

An LTL Verification System Based on Automata Theory

Cornelia van Wyk



THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH

Promoter: P. J. A. de Villiers
December 1999

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Opsomming

'n Stelsel vir die ontwerp en verifikasie van reaktiewe stelsels is by die Universiteit van Stellenbosch ontwikkel. Modeltoetsing word gebruik om korrektheidseienskappe, uitgedruk in CTL ('n vertakkende temporale logika), te toets. Die stelsel wat verifieer moet word, word modelleer in die hoëvlak spesifikasietaal ESML.

Die tesis beskryf die implementering van 'n LTL (lineêre tyd logika) modeltoetser vir ESML. Die nuwe modeltoetser is gebaseer op outomaatteorie, maar gebruik dieselfde toestandgenerasie tegniek vir ESML as die CTL modeltoetser. Die benadering wat gevolg is, is om LTL formules om te skakel in Büchi outomate voor die modeltoetsprosedure. Verifiëring geskied dan deur te toets of die produk van die Büchi outomaat en toestanddiagram van die ESML model leeg is.

Die algoritmes om Büchi outomate vanaf LTL formules te bou, die toestandgenerasie tegniek van die modeltoetser, en die algoritme om die produk van 'n Büchi outomaat en toestanddiagram te bereken, word gegee. Evaluasie van die nuwe modeltoetser het toetsing en vergelykings met SPIN en die CTL modeltoetser ingesluit. 'n Aantal effektiwiteitskwessies word bespreek.

Abstract

A tool for the design and verification of reactive systems has been developed at the University of Stellenbosch. On-the-fly model checking is used to check correctness properties expressed in CTL (Computation Tree Logic). The system to be verified is modelled in a specification language called ESML.

This thesis describes the implementation of an LTL (Linear Time Logic) model checker for ESML. The new model checker is based on automata theory, but uses the same state generator as the CTL model checker. The approach taken is to translate LTL formulas to Büchi automata before the model checking procedure. Verification proceeds by checking the emptiness of the product of the Büchi automaton and state graph generated from the ESML model.

The algorithms needed to build the Büchi automaton from an LTL formula, the state generation strategy used in the model checker, and the algorithm to compute the product of the state graph and Büchi automaton are given. Evaluation of the new model checker involved testing and comparison against SPIN and the CTL model checker for ESML. Some efficiency issues are discussed.

Vir my pa
wat my inspireer het om so ver te studeer;
En my ma
wat altyd help keer dat ek die lewe te ingewikkeld maak.

Contents

1	Introduction	1
1.1	Thesis outline	2
2	Background and Theory	4
2.1	The linear time temporal logic LTL	5
2.1.1	Syntax and semantics of LTL	5
2.1.2	Specifying properties in LTL	6
2.2	Automata on infinite words	8
2.2.1	Büchi automata	8
2.3	LTL model checking based on automata theory	10
3	Translating LTL Formulas into Büchi Automata	11
3.1	The translation process	12
3.2	Expanding the formula into a Nodeset	13
3.2.1	The data structures	15
3.2.2	The expansion algorithm	16
3.3	Building the Büchi automaton from the Nodeset	19
3.4	Algorithms to generate a Büchi automaton	26

3.4.1	Building the Büchi automaton	27
3.4.2	Computing the level of a target node	29
3.5	A final example	30
4	The model checker and its components	33
4.1	ESML	33
4.2	The state generator	35
4.3	The analyser	36
4.4	The iterative version of the model checking algorithm	38
5	Evaluation and conclusions	42
5.1	LTL to Büchi automata translator	43
5.2	The model checking algorithm	45
5.3	Future work	47
5.4	General conclusions	49
A	Algorithms	50
	Bibliography	55

List of Tables

1	The intermediate nodes between s_0 and s_1 of Figure 4.	17
2	The Nodeset for $G((p \text{ U } q) \wedge (r \text{ U } s))$	21
3	The U formulas and R formulas for the nodes of the Nodeset in Table 2. . . .	27
4	The U formulas and R formulas for the nodes of the Nodeset in Figure 14. . .	31
5	The number of nodes in Büchi automata generated with three different translation algorithms.	44
6	Second search results. Columns 3 and 4 show the number of states visited during each search.	45

List of Figures

1	The Büchi automaton for $G(p \cup q)$	9
2	The Büchi automaton for $p \cup q$	9
3	The Nodeset and Büchi automaton for the LTL formula $p \cup q$	13
4	The Nodeset for the LTL formula $p \wedge q$	14
5	Expanding the LTL formula $p \cup q$	14
6	The Nodeset for the LTL formula $p \cup q$	15
7	The Nodeset and Büchi automaton for the LTL formula $p \cup q$	19
8	The optimized Büchi automaton for the LTL formula $p \cup q$	20
9	The graph representation of the Nodeset in Table 2.	21
10	The Büchi automaton using the nodes of Table 2 as states.	22
11	Inserting the first nodes in the Büchi automaton for $G((p \cup q) \wedge (r \cup s))$	24
12	Inserting the last nodes in the Büchi automaton for $G((p \cup q) \wedge (r \cup s))$	25
13	The Büchi automaton for LTL formula $G((p \cup q) \wedge (r \cup s))$	26
14	The Nodeset for the LTL formula $((p \cup q) \vee (r \cup s))$	31
15	The Büchi automaton for the LTL formula $((p \cup q) \vee (r \cup s))$	32
16	The optimized Büchi automaton for the LTL formula $((p \cup q) \vee (r \cup s))$	32
17	An ESML (left) and a Promela (right) model for the alternating bit protocol	43
18	Executing the nested depth-first search on a graph with one cycle.	46

19	The state graph of a mutual exclusion model.	47
20	The state graph of a mutual exclusion model with an error.	48

List of Listings

3.1	Building the Büchi automaton from the Nodeset.	28
3.2	Computing the level of a node in a Büchi automaton.	29
3.3	Computing the number of levels to skip.	30
4.1	An ESML model for the Alternating Bit Protocol.	34
4.2	The definitions of propositions p and q	35
4.3	The nested depth first search algorithm.	37
4.4	The outline of procedure Check.	38
4.5	The code that implements the reaction to AllChildrenExplored.	40
4.6	Backtracking in the state graph.	41
A.1	The expansion algorithm.	52
A.2	The procedure that runs the model checking algorithm.	54

Acknowledgements

I want to thank

- My supervisor, Pieter de Villiers, for his guidance and advice
- My family and friends for their support and prayers
- Andrew Inggs for being enthusiastic about my work and lending a critical ear when I needed one
- Willem Visser who was always more than willing to answer my questions
- The staff and students of the Department of Computer Science, specifically: Reg Dodds, Jaco Geldenhuys, Tony Krzesinski, Abriette Senekal and Lynette van Zijl
- Telkom for their financial support

Chapter 1

Introduction

Computer-controlled systems have become extremely important in our everyday lives. Not only are they used in automated systems such as factories, but also life-critical systems in hospitals and aeroplanes where people's lives are at stake. Generally these systems are reactive, which means they maintain an ongoing interaction with their environment. Except in the most trivial cases, reactive systems comprise several concurrent processes. The complexity of these systems often results in subtle errors that are difficult to detect. As a result the need to check the correctness of the designs of computer systems before they are implemented has increased in importance. A verification system is a tool that can be used to check that a system design meets its specification.

Temporal logic can express the ordering of events in time without introducing time explicitly and has been shown to be suitable for the specification of correctness properties of reactive systems [10, 20, 24]. Specification languages have been developed which can be used to specify the interaction between the processes of a reactive system. From such a specification all possible behaviours of the system can be generated by starting from the initial state and using reachability analysis. The result is a state graph, a directed graph in which the nodes are global states of the system and the edges are atomic state transitions between states. Generating a state graph is completely mechanical and can be automated.

A model checker is a verification system that automatically verifies that the state graph of a system is a model for a temporal logic formula specifying a correctness property of the system. Two kinds of temporal logic often used in model checkers are linear time temporal logic [30] and branching time temporal logic [5, 6, 25]. This thesis describes the modification

of an existing model checker which uses the branching time temporal logic CTL (Computation Tree Logic) to use the linear time temporal logic LTL (Linear Time Logic).

The CTL model checker was developed at the University of Stellenbosch. In this model checker, the system to be verified is expressed in the specification language ESML [8], based on CSP [15]. The modular design of the model checker made it easy to extend. To support the checking of LTL correctness properties, only the module which implements the model checking algorithm had to be replaced. An automata-theoretic approach was used, in which both the system to be verified and the LTL formula are modelled as automata. The state graph generated from the ESML model can be viewed as an automaton and the LTL formula is negated and translated into a special automaton called a Büchi automaton. The model checking algorithm proceeds by computing the synchronous product of the state graph and Büchi automaton; if it is empty, the ESML model satisfies its specification.

For the most part the thesis describes algorithms and techniques that are known in the literature and have previously been implemented by others. The unique contribution of this thesis is a clear and concise description of how to translate “Nodesets” (graphs that represent simple Büchi automata) to generalized Büchi automata, that although present in the SPIN source code, is not described anywhere, to the best of our knowledge.

1.1 Thesis outline

Chapter 2 presents an overview of the relevant theory. The syntax and semantics of LTL are defined, the theory of Büchi automata is introduced, and a brief description of automata-theoretic model checking is given.

Chapter 3 describes how the Büchi automaton for an LTL formula is constructed. The construction is based on an algorithm developed by Gerth, Peled, Vardi and Wolper [14] and algorithms in SPIN [16]. The algorithm in [14] is designed to generate Büchi automata on-the-fly. Here it is combined with techniques from SPIN to build the Büchi automaton before model checking starts.

In Chapter 4 modifications to the model checker are discussed. The compiler was modified to parse LTL formulas and to generate appropriate code. The translation algorithm for LTL formulas to Büchi automata was integrated with the rest of the system. A model checking algorithm, developed by Courcoubetis, Vardi, Wolper and Yannakakis [7], that computes the

CHAPTER 1. INTRODUCTION

3

product of the Büchi automaton and the state graph was implemented (and adapted for this system). In the last chapter, Chapter 5, conclusions are drawn and future work discussed.

Chapter 2

Background and Theory

Given a specification of a system, a state graph of all the reachable system states can be generated mechanically. Correctness properties of the system can be expressed using temporal logic. In a nutshell, a model checker can be described as a verification tool that checks whether a state graph of a system satisfies a given correctness property.

The model checker described in this thesis is based on automata theory. LTL is used to describe correctness claims and an on-the-fly model checking algorithm is used. An on-the-fly model checker computes the reachable states as needed and therefore only a part of the state graph has to be stored in memory [3, 12, 13, 32]. A given LTL formula specifies a desirable property that all computations of a system should have. A computation can be seen as an infinite sequence of states. In the state graph of a system, every state is described by a finite set of atomic propositions, so a computation can be viewed as an infinite word over the alphabet of truth assignments to the atomic propositions. To check that the computations of a system are accepted by the LTL specification an approach is taken based on the theory that temporal formulas can be associated with finite state automata. Branching time temporal logic formulas correspond to automata on infinite trees [23] and alternating automata represent these formulas effectively [2, 18, 33]. Linear time temporal logic formulas are associated with automata on infinite words [29, 30, 34] and specifically, Büchi automata can be used to represent these formulas effectively [14, 31].

SPIN [16] is a well-known example of an on-the-fly LTL model checker that is based on automata theory. The system to be verified is expressed in the specification language Promela. Correctness properties are specified as LTL formulas which are automatically negated and

converted into Büchi automata which express unacceptable behaviour. A Promela model is translated into a C program and extended with state space searching modules. This program is compiled and executed to perform the required verification. The product of the Büchi automaton and state graph of the Promela model is generated on-the-fly using a nested depth first search.

The rest of this chapter provides an overview of essential theoretical concepts.

2.1 The linear time temporal logic LTL

Temporal logic can express the ordering of events in time without introducing time explicitly and has been shown to be suitable for specifying correctness properties of reactive systems. Linear time temporal logic is a kind of temporal logic that is concerned with the logical properties of single execution paths of a system.

In this thesis, properties are expressed in the linear time temporal logic LTL (Linear Time Logic). For a thorough logical treatment the reader is referred to [9], but for completeness the syntax and semantics of LTL are given here. The exposition in the next section has been adopted from [14].

2.1.1 Syntax and semantics of LTL

LTL formulas are constructed from a set of atomic propositions, the standard Boolean operators, the unary temporal operator X and the binary temporal operator U . More precisely, given a finite set of propositions P , formulas are defined inductively as follows:

- every member of P is a formula,
- if ϕ and ψ are formulas, then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $X\phi$ and $\phi U \psi$.

An interpretation for a linear time temporal logic formula is an infinite word $\xi = x_0, x_1 \dots$ over the alphabet 2^P , i.e., a mapping from the naturals to 2^P . The elements of 2^P are interpreted as assigning truth values to the elements of P : elements in the set are assigned true, elements not in the set are assigned false. The suffix of ξ starting at x_i is written as ξ_i and $\xi \models \phi$ denotes: ξ satisfies formula ϕ . The semantics of LTL can then be defined as

- $\xi \models q$ if $q \in x_0$, for $q \in P$,
- $\xi \models \neg\phi$ if not $\xi \models \phi$,
- $\xi \models \phi \wedge \psi$ if $\xi \models \phi$ and $\xi \models \psi$,
- $\xi \models \phi \vee \psi$ if $\xi \models \phi$ or $\xi \models \psi$,
- $\xi \models X\phi$ if $\xi_1 \models \phi$,
- $\xi \models \phi U \psi$ if there is an $i \geq 0$ such that $\xi_i \models \psi$ and $\xi_j \models \phi$ for all $0 \leq j < i$.

The following abbreviations will be used:

- $\text{true} = p \vee \neg p$.
- $\text{false} = \neg \text{true}$.
- $F\phi = (\text{true} U \phi)$; ϕ holds in future (finally ϕ).
- $G\phi = \neg F\neg\phi$; ϕ holds globally (globally ϕ).
- $p \Rightarrow q = (\neg p \vee q)$

The temporal operator \vee is defined as the dual of U : $\phi \vee \psi = \neg(\neg\phi U \neg\psi)$ and is referred to as the release operator. The U operator is a strong until, since $\phi U \psi$ specifies that ψ must eventually become true.

Temporal operators can be characterized by fixed points of appropriate functionals. The definition of a fixpoint is given here as defined in [1]. Let A be a finite set, \geq a partial ordering on A and f a function: $A \mapsto A$. An element $x \in A$ is called a fixpoint of $f \iff f(x) = x$. An element $x \in A$ is called a least fixpoint of $f \iff f(x) = x \wedge (\forall y : y \in A : f(y) = y \Rightarrow x \geq y)$. If the least fixpoint of f exists, it is denoted by $\mu x.f(x)$.

2.1.2 Specifying properties in LTL

The correctness properties of a system that can be specified in LTL can be divided into two main types of properties [19, 21]:

- Safety properties: A safety property is one which states that something “bad” will never happen.
- Liveness properties (progress properties [21]): A liveness property is one which states that something “good” will eventually happen.

Safety properties are specified by formulas of the form Gp . This formula claims that all the states of a computation satisfy p . Liveness properties state that a program will eventually enter a desirable state. For example, Fp is a liveness property which claims that p is true at least once on a given path and GFp claims that p is true infinitely many times on a given path.

Another kind of property that can be expressed in LTL is fairness requirements [11]. Fairness is a restriction placed on the set of accepted execution paths of a system. Different forms of fairness have been identified and three main types as described by Emerson and Lei in [11] are given here. In the examples given below p_i denotes transition i is enabled and q_i denotes transition i is executed.

- Unconditional fairness (or impartiality): An execution path is unconditionally fair if all the transitions are executed infinitely often. A path is impartial if GFq_i holds for all $0 < i \leq n$, where n is the number of transitions on the path.
- Weak fairness: In a weakly fair execution path every transition enabled continuously is executed infinitely often. A path is weakly fair if $FGp_i \Rightarrow GFq_i$ holds for all $0 < i \leq n$, where n is the number of transitions on the path.
- Strong fairness: Execution paths are strongly fair if every transition that is enabled infinitely often is executed infinitely often. A path is strongly fair if $GFp_i \Rightarrow GFq_i$ holds for all $0 < i \leq n$, where n is the number of transitions on the path.

Fairness can be implemented in a model checker and is a desirable feature. If it is not implemented in a model checker, and systems with fairness requirements are checked, fairness has to be expressed as temporal formulas.

The kind of liveness and safety properties that should be checked depends on the system being verified. To verify the correctness of a system with respect to a specified property, a model checker has to check that the system, expressed as a finite-state graph, satisfies the specified

LTL formula. Given any LTL formula, one can construct a finite automaton on infinite words that accepts precisely the computations that satisfy the formula [14, 31].

2.2 Automata on infinite words

Büchi, McNaughton and Rabin developed a framework for automata on infinite objects in the sixties [4, 22, 26]. This led to research on the connection between finite automata on infinite words and temporal logic [27, 29]. In particular, it was shown that LTL formulas can be represented by Büchi automata [14, 30, 31]. In the next section the definition of Büchi automata is given as it was defined in [14, 28].

2.2.1 Büchi automata

Büchi automata are nondeterministic finite automata equipped with an acceptance condition that is appropriate for infinite words $\omega = a_0, a_1, \dots$. An ω -word is accepted if the automaton can read it from left to right while visiting a sequence of states in which some accepting state occurs infinitely often.

A simple Büchi automaton over the alphabet Σ is of the form $A = (\Sigma, Q, q_0, \rho, F)$ with a finite set of states Q , initial state q_0 , transition relation $\rho \subseteq Q \times \Sigma \times Q$, and a set $F \subseteq Q$ of accepting states. A run of A on an ω -word $\alpha = \alpha_0, \alpha_1, \dots$ is a sequence $s = s_0, s_1, \dots$ where $s_0 = q_0$ and $s_{i+1} \in \rho(s_i, \alpha_i)$, for all $i \geq 0$. A accepts α if some state of F occurs infinitely often in a run of A on α . Let $L(A) = \{s \in \Sigma^\omega \mid A \text{ accepts } s\}$ be the ω -language recognized by A . If $L = L(A)$ for some Büchi automaton A , L is said to be Büchi recognizable.

Simple and generalized Büchi automata only differ in their acceptance condition. In a generalized Büchi automaton F denotes a set of sets of accepting states and a run of a generalized Büchi automaton is called successful if for each acceptance set $F_j \in F$, there exists at least one state $s_i \in F_j$ that occurs infinitely often in s .

An example of a simple Büchi automaton is given here to illustrate the basic concepts of Büchi automata. A Büchi automaton with $\Sigma = \{p, q\}$, $Q = \{s_0, s_1\}$, $q_0 = s_0$, $F = \{s_1\}$, $\rho(s_0, p) = s_0$, $\rho(s_1, p) = s_0$, $\rho(s_1, q) = s_0$ and $\rho(s_0, q) = s_1$, will accept the ω -word $(p^*q)^\omega$ and is shown in Figure 1. This Büchi automaton accepts the same set of infinite words that satisfies the LTL formula $G(p \cup q)$.

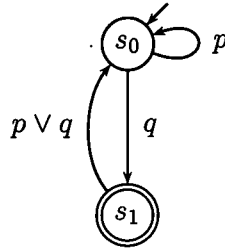


Figure 1: The Büchi automaton for $G(p U q)$.

Figure 1 illustrates the way Büchi automata will be represented in this thesis. In the representations of Büchi automata the initial state is indicated with a small arrow pointing towards it (in Figure 1 s_0 is an initial state) and an accepting state is indicated with a double circle (in Figure 1 s_1 is an accepting state). Transitions are labelled with the propositions that must be true to make a move from the origin state of the transition arrow to the target state of the transition arrow.

Consider the LTL formula $(p U q)$. This formula accepts the infinite word where p is true at all the states on a path until q is true. Once a state is reached where q is true, the formula is true independent of what the rest of the word looks like. The corresponding Büchi automaton is shown in Figure 2. The accepting state has a self loop labelled true, which means that once this accepting state is reached the automaton trivially accepts the input word. In the rest of the thesis, the true loop at such an accepting state will not be shown explicitly.

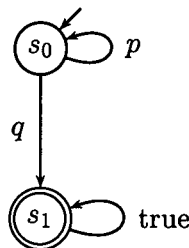


Figure 2: The Büchi automaton for $p U q$.

Several algorithms have been developed to translate an LTL formula to its corresponding Büchi automaton [14, 31]. As a result a model checker can use the Büchi automaton instead of the LTL formula during model checking.

2.3 LTL model checking based on automata theory

A model checker based on automata theory still solves the basic model checking problem. Given a program specification and an LTL formula, the model checker should check that all the computations of the program satisfy the LTL formula.

Let P denote a program and f an LTL formula. A Büchi automaton A_f that accepts exactly the computations that satisfy the formula f , is built. A finite-state graph that represents all the computations of program P is built. The finite-state graph is a Kripke structure $K_P = (\Pi, Q, R, q_0, \lambda)$ where Π is the set of propositions, Q is the set of states, $R \subseteq Q \times Q$ is the transition relation, q_0 is the initial state and $\lambda \in Q \mapsto 2^\Pi$ is the labelling. K_P can be viewed as a Büchi automaton $A_P = (\Sigma, Q, \rho, q_0, Q)$, where $\Sigma = 2^\Pi$ and $q' = \rho(q, a) \iff (q, q') \in R$ and $a \in \lambda(q)$. The automaton A_P has as its accepting set all the states in the automaton and therefore any run of the automaton is accepting. Let $L_\omega(A_P)$ be the set of all infinite words accepted by A_P and $L_\omega(A_f)$ the set of all infinite words accepted by A_f .

The verification process needs to check that all infinite words accepted by A_P are accepted by A_f . Since $L_\omega(A_f)$ is the set of all infinite words accepted by A_f , the verification process needs to check that $L_\omega(A_P) \subseteq L_\omega(A_f)$. Equivalently (if $\neg A_f$ = the complement of A_f) the verification process needs to check that the automaton that accepts $L_\omega(A_P) \cap L_\omega(\neg A_f)$ is empty, but complementing Büchi automata is very inefficient [27]. In [29] it is shown that $L_\omega(\neg A_f) = L_\omega(A_{\neg f})$, and since $L_\omega(A_P) \cap L_\omega(A_{\neg f}) = L_\omega(A_P \times A_{\neg f})$ the problem is reduced to showing that $L_\omega(A_P \times A_{\neg f}) = \emptyset$. To solve this problem Courcoubetis, Vardi, Wolper and Yannakakis [7] identified the following steps:

1. Build the finite automaton on infinite words of the negation of the formula f . The resulting automaton is $A_{\neg f}$.
2. Compute the global behaviour of the program P by building A_P as defined above.
3. Take the product of A_P and $A_{\neg f}$.
4. Check whether the language of the product automaton is non-empty.

Steps 2, 3 and 4 are done on-the-fly and if the product automaton is non-empty, program P does not satisfy formula f and the word accepted by the product automaton is the counter example.

Chapter 3

Translating LTL Formulas into Büchi Automata

The previous chapter presented some background on LTL and Büchi automata. In this chapter algorithms are described for translating LTL formulas into Büchi automata.

Büchi automata are constructed before the model checking procedure. Building the Büchi automaton beforehand is acceptable, because formulas for specifying correctness claims are usually simple. The advantage of building the Büchi automaton beforehand is that the automaton can be reduced by removing duplicate states before model checking starts.

The translator is based on an algorithm developed by Gerth, Peled, Vardi and Wolper (GPVW) [14] and algorithms implemented in SPIN [16]. The GPVW algorithm is used in the first part of the translator to expand the formula into a set of nodes and transitions which is called a Nodeset. The Nodeset represents a generalized Büchi automaton and can be viewed as a graph. A generalized Büchi automaton has a set of sets of accepting states and a simple Büchi automaton has a single set of accepting states as defined in Chapter 2. Algorithms from SPIN are used in the second part of the translator that builds a simple Büchi automaton from the Nodeset and optimizes it by removing duplicate states.

3.1 The translation process

The translation from an LTL formula into a Büchi automaton can be divided into several steps.

1. In the first step the operators G and F in the original formula are replaced by their corresponding U and \forall (dual of U) formulas. Definitions are used to put the formula in a form where negations only precede propositional variables. For example $\neg(p \Rightarrow q)$ is replaced by $p \wedge (\neg q)$.
2. In the next step the formula is expanded into a Nodeset. The translator starts with an initial node at which the original formula that must be expanded is stored. The fixpoint definition of this formula is used to create successor nodes. The fixpoint definition of $p U q$, for example, is $p U q = q \vee (p \wedge X(p U q))$. When the main operator of the fixpoint definition is any of the connectives \vee , \forall or U , two successor nodes are created, otherwise only one successor node is created. The algorithm expands each new node created in the same way as the initial node, adds expanded nodes to the Nodeset, and stops when no more nodes are created.
3. The Nodeset is then translated into a simple Büchi automaton. The Büchi automaton is divided into levels based on the number of U subformulas that must be true to satisfy the original formula. The nodes of the Nodeset are then inserted into the levels of the Büchi automaton based on the set of subformulas that is true at each node and the level of its predecessor.
4. Finally the Büchi automaton is optimized by removing duplicate states and removing unreachable states. A state is considered to be a duplicate of another state if both states have similarly labelled outgoing arcs and are either both accepting or both not accepting. A duplicate state is removed by replacing all its incoming arcs by arcs to the state it duplicates and then deleting it. The process is repeated until no more duplicates exist.

Step 1 prepares the LTL formula to be in the form required by the GPVW algorithm. This algorithm is used to perform step 2 and is explained in more detail in section 3.2. The algorithm was slightly adopted to build Büchi automata before the model checking procedure, but otherwise stays the same. Steps 3 and 4 are executed using algorithms from SPIN, as they are used there, and are described in more detail in section 3.3.

As an example, the Nodeset of $p \cup q$ after steps 1 and 2 is shown in Figure 3a and its optimized Büchi automaton after steps 3 and 4 is shown in Figure 3b. In a Nodeset nodes instead of transitions are labelled and accepting nodes are not yet identified.

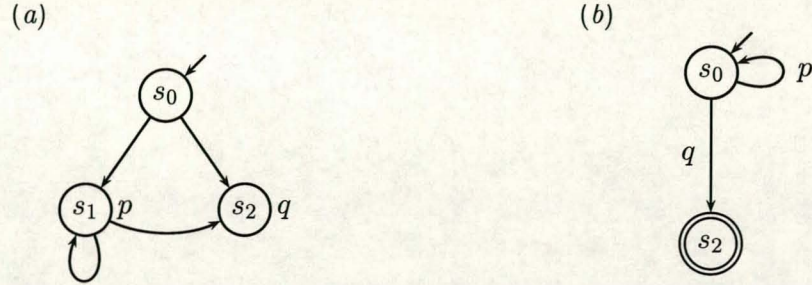


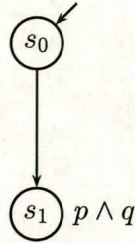
Figure 3: The Nodeset and Büchi automaton for the LTL formula $p \cup q$.

On an abstract level, the correspondence between the Büchi automaton in Figure 3b and the LTL formula $p \cup q$ is that the Büchi automaton accepts exactly the same set of infinite sequences of p 's and q 's that satisfy $p \cup q$. From the initial node s_0 it can either immediately move to node s_2 by reading a q or loop back to s_0 a finite number of times by reading a finite number of p 's and then move to node s_2 by reading a q . It will not accept a word with infinitely many p 's, because in this case it will never make a move to the accepting state s_2 .

The translator, however, operates on a more detailed level and therefore uses the correspondence between the building blocks of a Büchi automaton (nodes and transitions) and the building blocks of an LTL formula (subformulas) during the translation of an LTL formula into a Büchi automaton.

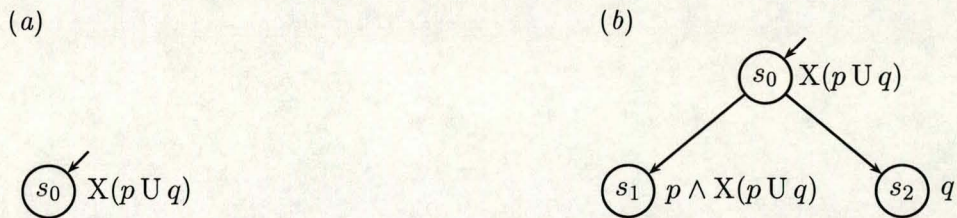
3.2 Expanding the formula into a Nodeset

The nodes of the Nodeset represent the subformulas of the original formula. Nodes s_1 and s_2 in Figure 3a, for example, represent the subformulas p and q of the formula $p \cup q$. However, generating a node for each subformula of the original formula results in an exponential blow-up of the size of the automaton [31]. In the Nodesets built with the GPVW algorithm some nodes can represent more than one subformula. For example, although $p \wedge q$ has two subformulas, namely p and q , one node can be used to represent $p \wedge q$ as shown in Figure 4.

Figure 4: The Nodeset for the LTL formula $p \wedge q$.

Individual nodes can represent subformulas without temporal operators, but subformulas with temporal operators are represented by sequences of nodes. Transitions between the nodes in such a sequence are inserted to represent the correct order of the nodes. For example, in Figure 3a nodes s_1 and s_2 , together with the loop from s_1 to s_1 , and the transition from s_1 to s_2 represent $p \cup q$.

To expand a formula with temporal operators into a sequence of nodes, its fixpoint definition is used. The fixpoint definition of $p \cup q$, for example, is $p \cup q = q \vee (p \wedge X(p \cup q))$. That is: if p is true at the current node then $p \cup q$ must be true at the next node, but if q is true, no restriction is placed on what must be true at the next node.

Figure 5: Expanding the LTL formula $p \cup q$.

The LTL formula $p \cup q$ is expanded by starting at an initial node (s_0 in Figure 5a), where $p \cup q$ must be true at all immediate successor nodes. Because $p \cup q$ must be true at the next node, successors for s_0 must be created. According to the fixpoint definition of $p \cup q$, either q must be true or $p \wedge X(p \cup q)$ must be true which means two successors must be created. The successor that represents q will be a single node, s_2 in Figure 5b. Subformula $p \wedge X(p \cup q)$,

however, has temporal operators (specifically it has a subformula that must be true at the next node) and therefore the successor that represents it (node s_1 in Figure 5b), needs further expansion.

At node s_1 in Figure 5b, p is true and $p \cup q$ must be true at the next node. Again, according to the fixpoint definition of $p \cup q$, two successors are created for s_1 . A single successor (node s_4 in Figure 6a) is created to represent q , but an identical node already exist, namely s_2 . Therefore, instead of inserting s_4 , a transition is inserted from node s_1 to node s_2 (see Figure 6b). The other successor, node s_3 in Figure 6a, represents $p \wedge X(p \cup q)$, but an identical node also already exists. Therefore, instead of creating node s_3 , a transition is inserted from node s_1 to node s_1 (see Figure 6b). All the nodes have now been expanded and the graph in Figure 6b is the Nodeset for $p \cup q$.

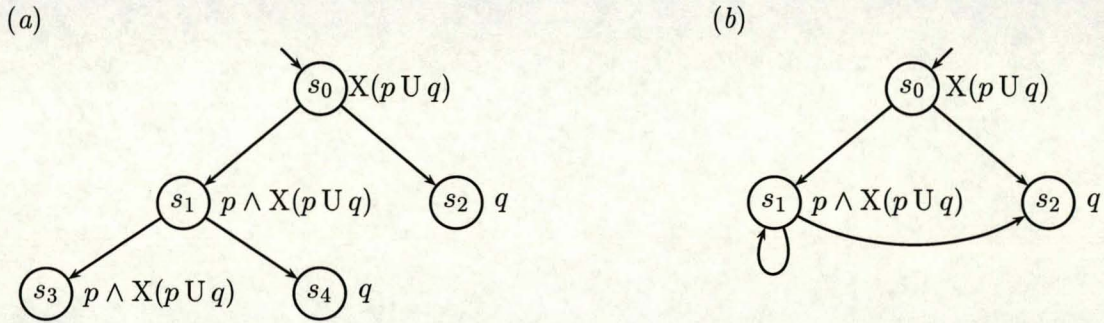


Figure 6: The Nodeset for the LTL formula $p \cup q$.

In principle, the GPVW algorithm follows the same basic steps. However, it operates on a more detailed level and a number of intermediate nodes are created when a node is expanded to create successors. At the intermediate nodes the formula that is expanded is in different stages of expansion.

In the rest of this section the GPVW algorithm is described. The complete algorithm is given in Appendix A.

3.2.1 The data structures

The data structure that represents a node has the following fields:

- Field New stores formulas during their different phases of expansion.

- Field TFormulas stores the formulas that are true at the current node and the formulas that must be true on the current path (the formulas that were expanded on the path to the current node).
- Field Next stores the formulas that must be true at the next node.
- Fields Incoming and Outgoing store the names of nodes at the origin and target of the node's incoming and outgoing transitions respectively.

The Nodeset is one linked list of nodes. Only nodes that are fully expanded are added to the Nodeset. When a node is fully expanded field New is empty, because New only stores formulas during their different phases of expansion. The data structure of the Nodestack, a stack that is used to store intermediate nodes during their different phases of expansion, is also a linked list of nodes.

The data structures are defined in lines 1–16 of Listing A.1 in Appendix A.

3.2.2 The expansion algorithm

The algorithm starts with an initial node (with the original formula stored in field New) which is pushed on the Nodestack. This node is then expanded to create its successors. Each newly created node is pushed on the stack. One node at a time is popped from this stack and expanded (with procedure Expand), until the stack is empty. This is illustrated by:

```

WHILE Nodestack # empty DO
  tmp := PopOffStack(Nodestack);
  Expand(tmp)
END

```

As an example, the sequence of intermediate nodes between node s_0 and s_1 of the Nodeset in Figure 4 is shown in Table 1. This sequence of nodes is created when the LTL formula $p \wedge q$ is expanded into a Nodeset.

s'_1 is the first node that is pushed on the stack. This node is then popped and expanded and as a result s''_1 is pushed on the stack. s''_1 is then popped and expanded and as a result s'''_1 is pushed on the stack. When the last node in the table (s''''_1) is popped, it is added to the Nodeset as node s_1 , because it is fully expanded. s_0 is stored in the Incoming field of s_1 and

Node	Incoming	New	TFormulas	Next
s'_1	s_0	$p \wedge q$	\emptyset	\emptyset
s''_1	s_0	p, q	\emptyset	\emptyset
s'''_1	s_0	q	p	\emptyset
s''''_1	s_0	\emptyset	$p \wedge q$	\emptyset

Table 1: The intermediate nodes between s_0 and s_1 of Figure 4.

therefore s_1 is the successor of s_0 . Since node s_1 has no formula that must be true at the next node, no successor nodes are created for it and the final Nodeset has only two nodes as shown in Figure 4. The actions of Procedure Expand are shown in the pseudo-code that follows.

```

IF Node.New = empty THEN
  IF no duplicate of Node in Nodeset THEN Add Node to Nodeset
  ELSE Add Node.Incoming to field Incoming of the duplicate END;
  IF Node.Next  $\neq$  empty THEN Create successor node and push it on Nodestack END
ELSE
  Expand Node to create successor(s);
  Push successor(s) on the Nodestack
END

```

When procedure Expand is called to expand a node y and y is fully expanded (y .New = empty), a successor of one of the nodes in the Nodeset has been reached. If an identical node to y , called x , is already in the Nodeset, y .Incoming is added to x .Incoming, otherwise, y is added to the Nodeset. If node y has a formula that must be true at the Next node (field Next is not empty), an intermediate node that will be expanded to create y 's successors is pushed on the Nodestack.

When procedure Expand is called to expand a node y , and y is not fully expanded (y is not empty), it is an intermediate node that needs further expansion. A formula, ϕ , is then removed from y .New and successor nodes are created based on the fixpoint definition of ϕ : if the main operator of ϕ is any of the connectives \vee , \cup or \bigvee , two nodes are created and pushed on the stack, otherwise only one node is created and pushed on the stack.

- If ϕ is a proposition, node y is again pushed on the stack, but with ϕ added to TFormulas.

- If $\phi = p \vee q$, $\phi = p \cup q$ or $\phi = p \vee q$, two new nodes, a and b , are created and pushed on the Nodestack. Field Incoming of both nodes a and b is assigned the values in field Incoming of node y , because they are intermediate nodes between a predecessor and its two successors respectively. Field TFormulas of both nodes a and b are assigned the union of y .TFormulas and ϕ .

Fields New and Next of a and b , are assigned values as follows:

- If $\phi = p \vee q$, p is added to field New of one copy and q to field New of the other copy, unless they are already members of TFormulas. The reason is that when a formula is a member of TFormulas, it means that it is already expanded on the current path and does not need to be expanded again.

$a.\text{New} := y.\text{New} \cup (\{p\} \setminus y.\text{TFormulas});$

$a.\text{Next} := y.\text{Next};$

$b.\text{New} := y.\text{New} \cup (\{q\} \setminus y.\text{TFormulas});$

$b.\text{Next} := y.\text{Next};$

- If $\phi = p \cup q$, p is added to field New (unless it is already a member of TFormulas) and $p \cup q$ to field Next of one copy and q is added to field New (unless it is already a member of TFormulas) of the other copy. These actions are based on the fixpoint definition $p \cup q = q \vee (p \wedge X(p \cup q))$. The code used to execute these assignments is shown below:

$a.\text{New} := y.\text{New} \cup (\{p\} \setminus y.\text{TFormulas});$

$a.\text{Next} := y.\text{Next} \cup \{(p \cup q)\};$

$b.\text{New} := y.\text{New} \cup (\{q\} \setminus y.\text{TFormulas});$

$b.\text{Next} := y.\text{Next};$

- If $\phi = p \vee q$, q is added to field New (unless it is already a member of TFormulas) of both nodes and p is added to the field New (unless it is already a member of TFormulas) of one copy and $p \vee q$ to the Next of the other copy. These actions are based on the fixpoint definition $p \vee q = q \wedge (p \vee X(p \vee q))$. The code used to implement this is shown below:

$a.\text{New} := y.\text{New} \cup (\{(p \wedge q)\} \setminus y.\text{TFormulas});$

$a.\text{Next} := y.\text{Next};$

$b.\text{New} := y.\text{New} \cup (\{q\} \setminus y.\text{TFormulas});$

$b.\text{Next} := y.\text{Next} \cup \{(p \vee q)\};$

- If $\phi = p \wedge q$, node y is again pushed on the stack, but with ϕ added to TFormulas and both p and q added to New (unless they are already members of TFormulas), because both must be true for ϕ to be true.

Procedure Expand is given in lines 22–81 of Listing A.1 in Appendix A.

After the last node has been popped from the Nodestack and the Nodeset has been computed, the set of outgoing transitions for each node is computed. It is done by using the Incoming fields of all the nodes. This prepares the Nodeset before translation into a Büchi automaton.

3.3 Building the Büchi automaton from the Nodeset

For simple formulas the Nodeset can be transformed into a simple Büchi automaton directly. For example, the Nodeset of $p \text{ U } q$ in Figure 7a can be transformed into its corresponding Büchi automaton (shown in Figure 8) by marking accepting states, labelling transitions and removing duplicate states.

Following these steps the Büchi automaton from the Nodeset in Figure 7a will be constructed first. Then another example will be discussed to show for which kind of formulas this direct transformation from a Nodeset (that represents a generalized Büchi automaton) to simple Büchi automaton does not work. Finally a translation process that correctly translates a Nodeset into a simple Büchi automaton will be given. This solution is implemented in SPIN [16], but has not been published, to the best of our knowledge.

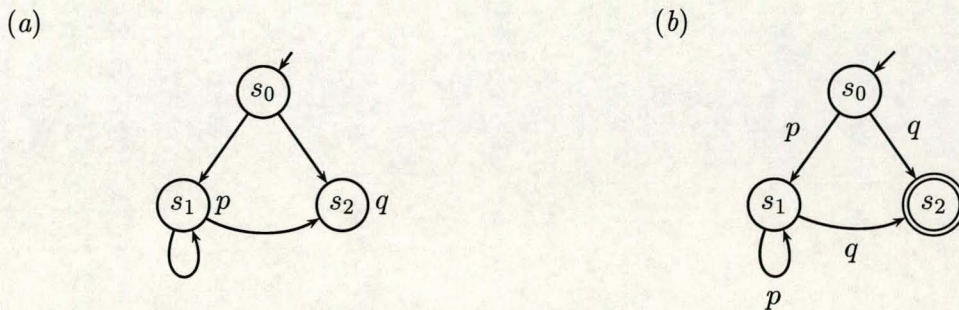


Figure 7: The Nodeset and Büchi automaton for the LTL formula $p \text{ U } q$.

The first step in the direct transformation of the Nodeset of $p \text{ U } q$ into a Büchi automaton is

to mark the acceptance states. A state can be marked as accepting if any path from the initial state to that state produces a sequence of subformulas that satisfies the original formula at least once. For Gp for example, a state will be marked accepting if p is true at least once on every path from the initial state to this state. Of course this state must be reachable from itself. Such a state will be reachable from itself as a result of the GPVW algorithm. The expansion of a formula like Fp that does not require p to be true infinitely often will result in an accepting state with no outgoing arcs. In such a case the accepting state has an implicit self loop labelled true.

Node s_1 in Figure 7a is not accepting, because a path that starts at node s_0 and loops infinitely often through node s_1 does not satisfy $p \cup q$. Node s_2 , on the other hand, is accepting, because at this node q is true and any path from s_0 to s_2 produces a sequence of subformulas that satisfies $p \cup q$.

In a Nodeset nodes are labelled, but in a Büchi automaton transitions are labelled. The second step in the direct transformation of the Nodeset of $p \cup q$ into a Büchi automaton is to label transitions. A transition is labelled with the set of propositions that is true at the target node of that transition. After node s_2 in Figure 7a has been marked as accepting and the transitions have been labelled the resulting Büchi automaton for $p \cup q$ is the one shown in Figure 7b.

Finally, nodes s_0 and s_1 in Figure 7b are identified as duplicates, because they have the same outgoing transitions and are both not accepting. Duplicate state s_1 is removed by replacing all of its incoming arcs by arcs to the state it duplicates (namely s_0), and deleting it (see Figure 8).

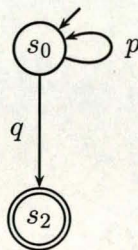


Figure 8: The optimized Büchi automaton for the LTL formula $p \cup q$.

By labelling transitions and removing duplicate states, the Nodeset in this example was

transformed into its corresponding simple Büchi automaton, but this method fails for some formulas. The LTL formula $G((p \cup q) \wedge (r \cup s))$ is such a formula. This formula, written in the form requested by the algorithm, is: $\text{false} \vee ((p \cup q) \wedge (r \cup s))$. The fixpoint definition of this formula is: $((q \vee (p \wedge X(p \cup q))) \wedge (s \vee (r \wedge X(r \cup s)))) \wedge X(\text{false} \vee ((p \cup q) \wedge (r \cup s)))$. Expanding the formula into a set of nodes and transitions as described in the previous section the result is the Nodeset in Table 2. In this table the set of outgoing nodes and the set of formulas that are true at a node is given for each node. The graph representation of the Nodeset is shown in Figure 9.

Node	Outgoing Nodes	Formulas true at Node
s_0	s_1, s_2, s_3, s_4	
s_1	s_1, s_2, s_3, s_4	$p \wedge r$
s_2	s_1, s_2, s_3, s_4	$p \wedge s$
s_3	s_1, s_2, s_3, s_4	$q \wedge r$
s_4	s_1, s_2, s_3, s_4	$q \wedge s$

Table 2: The Nodeset for $G((p \cup q) \wedge (r \cup s))$.

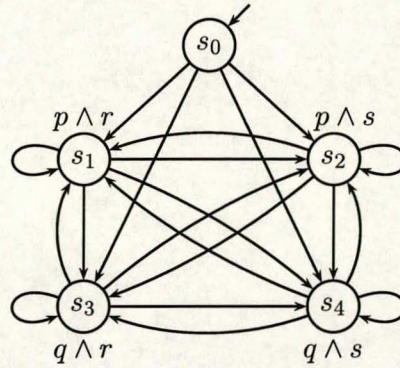


Figure 9: The graph representation of the Nodeset in Table 2.

Following the steps used in the previous example, node s_4 of this Nodeset is marked as accepting, because q and s are true at this node and satisfy the original formula. Again, transitions must be labelled instead of nodes. The transitions are labelled as follows:

- All the transitions with node s_1 as target, are labelled $p \wedge r$ and label $p \wedge r$ is removed from node s_1 .

- All the transitions with node s_2 as target, are labelled $p \wedge s$ and label $p \wedge s$ is removed from node s_2 .
- All the transitions with node s_3 as target, are labelled $q \wedge r$ and label $q \wedge r$ is removed from node s_3 .
- All the transitions with node s_4 as target, are labelled $q \wedge s$ and label $q \wedge s$ is removed from node s_4 .

Finally nodes s_1 , s_2 and s_3 are identified as duplicates of node s_0 , because none of them is accepting and they all have outgoing transitions to s_1 , s_2 , s_3 and s_4 . Nodes s_1 , s_2 and s_3 are deleted after all their incoming arcs are replaced by arcs to node s_0 . The three self loops created at node s_0 as a result of the deletion of its duplicates can then be replaced by one transition. This transition is labelled with the disjunction of the labels of the transitions it replaces. In a similar manner the three transitions from node s_4 to node s_0 , also created as a result of the deletion of node s_0 's duplicates, are replaced by one transition. The final Büchi automaton is shown in Figure 10.

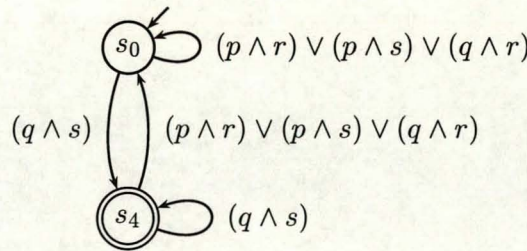


Figure 10: The Büchi automaton using the nodes of Table 2 as states.

This Büchi automaton accepts infinite paths on which $p \cup q$ and $r \cup s$ are both true infinitely often, with the added restriction that q and s must be true at the same states along the path infinitely often. $G((p \cup q) \wedge (r \cup s))$ on the other hand accepts infinite paths on which $p \cup q$ and $r \cup s$ are both true infinitely often, but q and s do not have to be true at the same states along the path infinitely often. For example the infinite word $((p \wedge r), (q \wedge r), (q \wedge r), (p \wedge s))^\omega$ will be accepted by the LTL formula $G((p \cup q) \wedge (r \cup s))$, but not by the automaton in Figure 10.

The transformation failed, because $G((p \cup q) \wedge (r \cup s))$ has two \cup subformulas that must be true to satisfy it and this information is lost with the direct transformation into a simple Büchi automaton described above.

A state in the Büchi automaton for $G((p \cup q) \wedge (r \cup s))$ will be accepting if both q and s are true at least once on every path from the initial state to that state. This does not require q and s to be true at the same state to satisfy $G((p \cup q) \wedge (r \cup s))$, as is the case with the automaton in Figure 10. The transformation does not fail for all formulas with more than one \cup subformula, for example: $G((p \cup q) \vee (r \cup s))$ has two subformulas, but only one of the two subformulas has to be true for the original formula to be true. A state in its corresponding Büchi automaton will therefore be marked accepting if, on every path from the initial state to that state, either q or s is true at least once.

In the rest of this section a method for building a Büchi automaton from a Nodeset without losing information, is described. Based on the \cup subformulas in the original formula, the states of the Büchi automaton are divided into levels; an initial level and then one level for each \cup subformula that must be true for the formula to be true. Each level, except the first level, is associated with a \cup subformula and is assigned an acceptance condition. The acceptance condition of a level is the set of subformulas that will satisfy the \cup subformula associated with that level and all the \cup subformulas of the lower levels. The acceptance condition of the highest level is the set of subformulas that must be true to satisfy the original formula. The highest level is called the acceptance level and nodes inserted in this level are marked as acceptance nodes.

For example, the Büchi automaton of $G((p \cup q) \wedge (r \cup s))$ has three levels, because it has two \cup subformulas that must be true to satisfy it. If $p \cup q$ is associated with the second level and $r \cup s$ with the third level, q is the acceptance condition of the second level and $q \wedge s$ is the acceptance condition of the third level. In this case the third level will be the acceptance level.

The Büchi automaton of an LTL formula with no \cup subformulas or just one \cup subformula that must be true to satisfy it, has two levels, the initial level and an acceptance level. The acceptance condition of the acceptance level is then the set of subformulas that must be true to satisfy the LTL formula.

During translation of a Nodeset into a Büchi automaton, a node is inserted into a level of the Büchi automaton based on the set of formulas that is true at the node and the level of its predecessor. When the transitions to successor nodes are inserted, they are labelled with the set of formulas that is true at each successor respectively.

The initial node of the Nodeset is inserted in the first level of the Büchi automaton, which

will be node s_0 for the Nodeset of $G((p \cup q) \wedge (r \cup s))$ in Table 2. This node has outgoing transitions to nodes s_1, s_2, s_3 and s_4 . The level into which each of these target nodes is inserted is computed using the set of formulas that is true at the target node and the fact that the predecessor node is in level 1. Because q is not true at nodes s_1 or s_2 , they cannot be inserted in level 2 or any level after level 2, and are therefore inserted in level 1. Node s_3 is inserted in level 2, because q is true at this node and node s_4 is inserted in level 3, because q and s are true at this node. The successor nodes are relabelled to s'_1, s'_2, s''_3 and s'''_4 . The Büchi automaton after insertion of the initial node and its four successors is shown in Figure 11a.

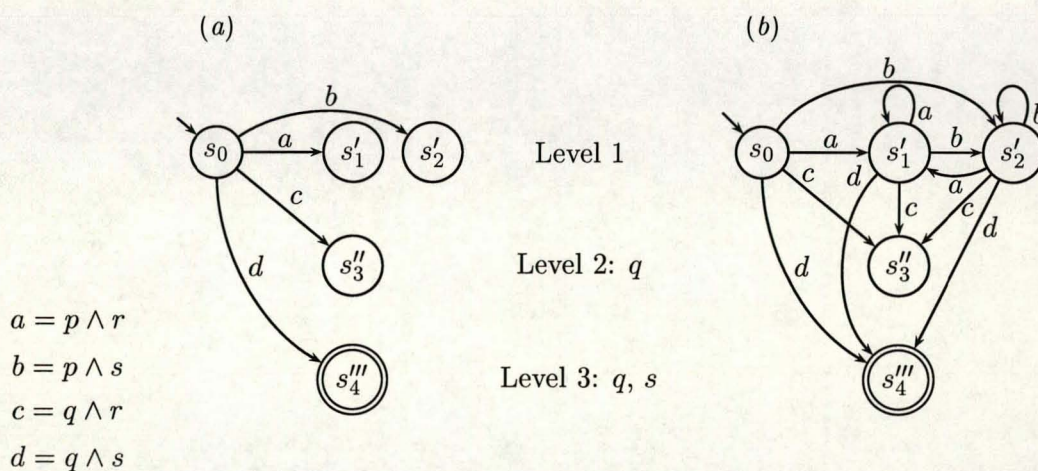


Figure 11: Inserting the first nodes in the Büchi automaton for $G((p \cup q) \wedge (r \cup s))$.

Node s_1 in Table 2 has outgoing transitions to nodes s_1, s_2, s_3 and s_4 . Node s'_1 is in level one (see Figure 11a) and its outgoing transitions are added with s'_1, s'_2, s''_3 and s'''_4 as target nodes (see Figure 11b). Node s_2 in Table 2 also has outgoing transitions to nodes s_1, s_2, s_3 and s_4 . Node s'_2 is in level one (see Figure 11a) and its outgoing transitions are therefore also added with s'_1, s'_2, s''_3 and s'''_4 as target nodes (see Figure 11b).

Node s_3 in Table 2 again has outgoing transitions to nodes s_1, s_2, s_3 and s_4 , but s''_3 in Figure 11a is in level 2. Its successor nodes will therefore be inserted in different levels to what they were inserted in as successors of nodes in level 1. The reason is that q is true at level 2.

The successors of s''_3 (see Figure 12a) are

- s'_1 in level 2, because q is true at its predecessor;

- s_2''' in level 3, because q is true at s_3'' and s is true at s_2''' , which means that both q and s are true at least once on every path from s_0 to s_2''' ;
- s_3'' in level 2, because q is true at s_3'' ;
- s_4''' in level 3, because q and s is true at s_4''' .

Similar to node s_3'' , s_1'' has transitions to nodes s_1' and s_3' in level 2, and transitions to nodes s_2''' and s_4''' in level 3 (see Figure 12b).

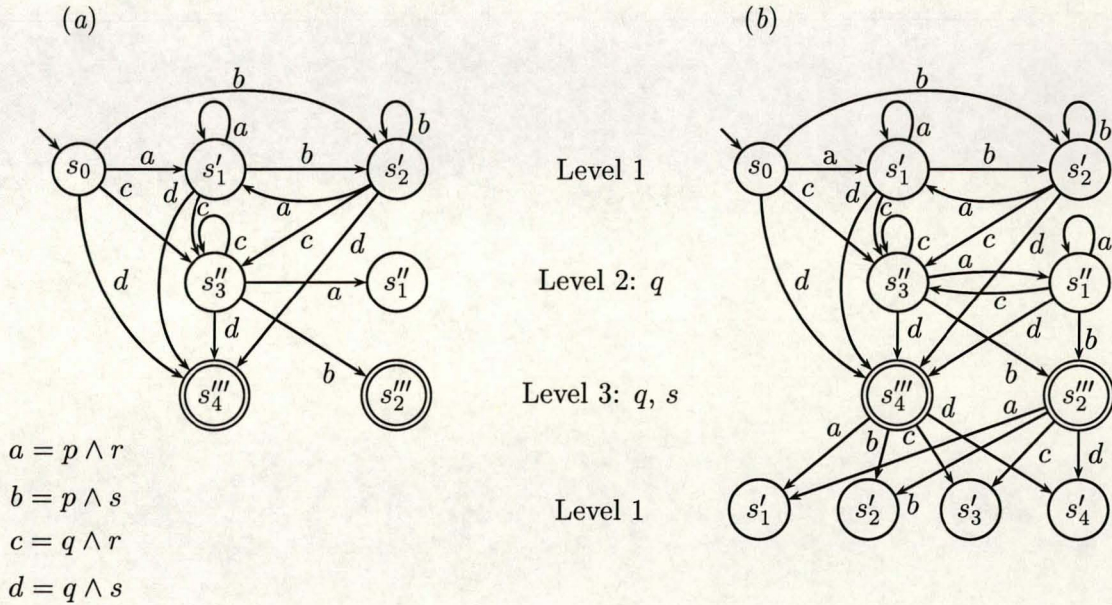


Figure 12: Inserting the last nodes in the Büchi automaton for $G((p \cup q) \wedge (r \cup s))$.

When all the paths from the initial node to an accepting node satisfy a formula that must be true infinitely often, at least once, that accepting state will have one or more outgoing transitions. These explicit outgoing transitions can therefore be added with nodes in the initial level as targets. If this method is followed for node s_4''' (in Table 2 s_4 has outgoing transitions to nodes s_1, s_2, s_3 and s_4), its outgoing transitions are added with s_1', s_2', s_3' and s_4' in level 1 as target nodes. Similarly to s_4''' , s_2''' has outgoing transitions to s_1', s_2', s_3' and s_4' . To represent the Büchi automaton after these steps (see Figure 12b), level 1 was duplicated below level 3. The two copies of node s_1' in Figure 12b are therefore exactly the same node. The target nodes of the four outgoing transitions of nodes s_3' and s_4' (not depicted in Figure 12b) are identical to the target nodes of the outgoing transitions of nodes s_0, s_1' and s_2' .

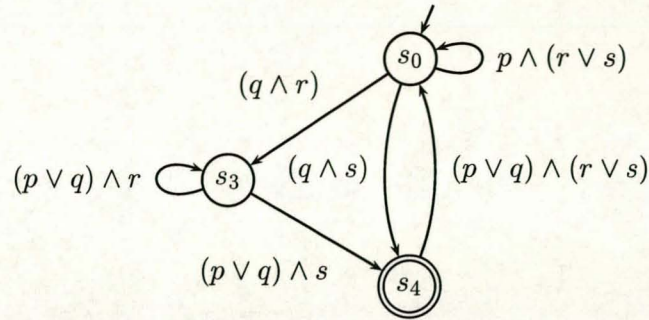


Figure 13: The Büchi automaton for LTL formula $G((p U q) \wedge (r U s))$.

If a Büchi automaton is built from a Nodeset as described above, no information will be lost with the deletion of duplicates. The translator proceeds by identifying nodes s'_1 , s'_2 , s'_3 and s'_4 as duplicates of node s_0 , nodes s''_1 as a duplicate of node s''_3 and node s'''_2 as a duplicate of node s'''_4 . After the duplicates have been removed, all the transitions that connect the same two nodes are replaced with one transition, labelled with the disjunction of the labels of the original transitions. The final Büchi automaton for the LTL formula $G((p U q) \wedge (r U s))$ has three nodes and is shown in Figure 13.

The algorithms which are used to build a simple Büchi automaton from a Nodeset, are described in the next section. These algorithms were adapted from the SPIN system's source code.

3.4 Algorithms to generate a Büchi automaton

In the previous section the number of levels in a Büchi automaton was computed by counting the number of U formulas *that must be true to satisfy the original formula*. The number of levels in the Büchi automaton generated with the translator is computed by just counting the number of U formulas in the original formula. If the number of U formulas in the original formula is more than the number of U formulas that must be true to satisfy the original formula, the extra levels will be redundant; all the nodes in these levels will be deleted during the optimization of the final Büchi automaton.

Subformulas that contain U operators are called U formulas and their right-hand side subformulas are called R formulas. For example: the R formula of $p \text{ U } q$ is q and the R formula of $p \text{ U } (r \text{ U } s)$ is $(r \text{ U } s)$. If there is more than one U formula in the original formula (as in the second example of the previous sentence), each U formula is assigned a unique number and each R formula is assigned the same number as the U formula in which it appears. The highest level number is equal to the number of levels in the Büchi automaton.

To assist the translator in computing the level number at which a node must be inserted, the numbers of all the U and R formulas found in TFormulas of each node are stored at that node. Field TFormulas of Nodes in the Nodeset, stores all the subformulas that must be true on the current path and all the subformulas that are true at the current node. So if the R formulas of all the U formulas in TFormulas are in TFormulas, the node can be inserted in the acceptance level. Two lists (called the Ulist and Rlist) are associated with each node and are used to store the numbers of U and R formulas (see line 14 of Listing A.1 in Appendix A). For each U subformula in TFormulas, its number is stored in the Ulist and for each R formula in TFormulas, its number is stored in the Rlist.

As an example consider the LTL formula $G((p \text{ U } q) \wedge (r \text{ U } s))$ with the two U formulas $(p \text{ U } q)$ and $(r \text{ U } s)$. Assign 1 to U formula $(p \text{ U } q)$ and 2 to U formula $(r \text{ U } s)$. Then R formula q is also assigned the number 1 and R formula s is assigned 2. Table 3 shows the numbers stored in each list for the nodes in the Nodeset. The second column shows the formulas in their TFormulas field and columns 3 and 4 show the numbers stored in the Ulist and Rlist respectively.

Nodes	TFormulas	U formulas	R formulas
1	$p \text{ U } q, r \text{ U } s, p, r$	1, 2	
2	$p \text{ U } q, r \text{ U } s, p, s$	1, 2	2
3	$p \text{ U } q, r \text{ U } s, q, r$	1, 2	1
4	$p \text{ U } q, r \text{ U } s, q, s$	1, 2	1, 2

Table 3: The U formulas and R formulas for the nodes of the Nodeset in Table 2.

3.4.1 Building the Büchi automaton

In the example of the previous section the initial node and its successors were inserted in the correct levels of the Büchi automaton and then for each successor its successors were inserted.

This process was repeated until the successors of all the nodes were added. The translator, however, inserts all the nodes of the Nodeset (and their outgoing transitions) in one level of the Büchi automaton at a time. The initial node is only inserted in the first level, but one copy of all the other nodes are inserted in each level. The algorithm that is used to insert all the nodes in each level is shown in Listing 3.1.

```

1  PROCEDURE CreateBüchi;
2  VAR
3    fromprefix: INT;
4  BEGIN
5    FOR i := 0 TO NumUntils DO
6      FOREACH node N in Nodeset DO
7        IF (i = 0) OR (N.Name # init) OR (NumUntils = 0) THEN
8          IF (i = NumUntils) AND (N.Name # init) THEN
9            fromprefix := Accept
10         ELSE
11           fromprefix := i
12         END;
13         InsertTransitions(N,i,fromprefix)
14       END
15     END
16 END CreateBüchi;

```

Listing 3.1: Building the Büchi automaton from the Nodeset.

Not all the nodes are needed in each level, but this mechanical procedure of inserting nodes is simpler and after the Büchi automaton has been built, unreachable nodes can be deleted. A prefix is added to the name of each node that is inserted in the Büchi automaton. This prefix stores the number of the level in which the node is inserted.

To insert a node in the Büchi automaton and add all its outgoing transitions, procedure InsertTransitions (line 13 of Listing 3.1) is called. When this procedure adds an outgoing transition to a target node with outgoing transitions it calls procedure GetNextLevel (Listing 3.2) to compute the level of the target node. When procedure InsertTransitions adds an outgoing transition to a target node with no outgoing transitions, it replaces the target node with a special node in the acceptance level, called “AcceptAll”. This node is an accepting state with an implicit self loop labelled true.

3.4.2 Computing the level of a target node

Procedure `GetNextLevel` in Listing 3.2 receives as input the target node of a transition and the level number of the current node (the node at the start of the transition). If the current node is in the acceptance level, the level of the target node is assigned the number of the initial level. If the original formula does not contain any U formulas, the Büchi automaton has only two levels; the level of the target node is assigned the number of the acceptance level. If no node in the Nodeset has any U or R formulas, the Büchi automaton also has only two levels and the level of the target node will again be assigned the number of the acceptance level. These actions are illustrated by lines 5–8 of Listing 3.2.

If none of these conditions apply, procedure `GetLevelincr` (Listing 3.3) is called to compute the number of levels to skip to get to the level of the target node (lines 10–15 of Listing 3.2). If this value is equal to or greater than the number of U formulas in the original formula, the acceptance conditions of all the levels have been met and the level of the target node is assigned the number of the acceptance level. Otherwise, the level of the target node is assigned the sum of the current level number and the number of levels to skip as computed by procedure `GetLevelincr`.

```

1  PROCEDURE GetNextLevel(currentlevel: INT; N: LTLNode): INT;
2  VAR
3    nextlevel, levelincr: INT;
4  BEGIN
5    IF (NumMarkedUntils = 0) OR (NumUntils = 0) THEN
6      nextlevel := lAccept goto acceptance level
7    ELSIF currentlevel ≥ NumUntils THEN
8      nextlevel := 0 Cycle to first level
9    ELSE
10     levelincr := GetLevelincr(N,currentlevel+1);
11     IF (levelincr + currentlevel) ≥ NumUntils THEN
12       nextlevel := lAccept last hop to acceptance level
13     ELSE
14       nextlevel := currentlevel + levelincr
15     END
16   END;
17   RETURN nextlevel
18 END GetNextLevel;
```

Listing 3.2: Computing the level of a node in a Büchi automaton.

Procedure `GetLevelincr` receives as input the target node and the level number of the next level after the level of the current node. Of course the target node will have duplicate states in each level and the aim of this algorithm is to find the node in the highest level to which a

transition can be added.

The algorithm investigates the numbers of the U formulas and R formulas in the Ulists and Rlists of the nodes to check if the subformulas that are true at the node satisfy the acceptance conditions of a higher level than the current level. If the acceptance conditions of a level is met, the algorithm repeats the test with the next level. This process continues until the last level is reached or the acceptance conditions of a level are not met.

If none of the acceptance conditions of any higher level could be met, a zero is returned for levelincr and the target node will have the same level number as its predecessor.

```

1  PROCEDURE GetLevelincr(Node: LTLNode; count: INT): INT;
2  VAR
3    levelincr: INT;
4  BEGIN
5    levelincr := 0;
6    WHILE count ≤ NumUntils DO                                check all levels
7      IF count in RList THEN                                    U formula true - can goto next level
8        levelincr := levelincr + 1;
9      ELSIF count in UList THEN                                U formula not true yet
10       RETURN levelincr
11    END
12    levelincr := levelincr + 1;                                U formula true - can goto next level;
13    count := count + 1
14  END;

16  RETURN levelincr                                           Number of levels to skip
17 END GetLevelincr;

```

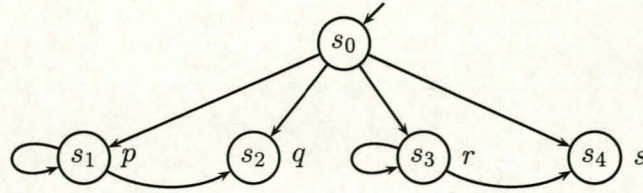
Listing 3.3: Computing the number of levels to skip.

Finally the Büchi automaton is optimized by removing duplicate states, deleting states that are not reachable, and replacing transitions that connect the same two nodes with one transition (labelled with the disjunction of the labels of the transitions it represents).

3.5 A final example

To demonstrate the second part of the translation process using the algorithms given in the previous section, the Nodeset of $((p \cup q) \vee (r \cup s))$ will be translated into a Büchi automaton. This formula has two U subformulas, but only one of them has to be true for the formula to be true. The graph representing the Nodeset for this formula is shown in Figure 14.

The Ulists and Rlists of the nodes in the Nodeset are shown in Table 4. Column 2 shows the

Figure 14: The Nodeset for the LTL formula $((p \text{ U } q) \vee (r \text{ U } s))$.

formulas in field TFormulas and columns 3 and 4 shows the numbers of the U formulas and R formulas in the Ulists and Rlists respectively.

Nodes	TFormulas	U formulas	R formulas
s_1	$p \text{ U } q, p$	1	
s_2	$p \text{ U } q, q$	1	1
s_3	$r \text{ U } s, r$	2	
s_4	$r \text{ U } s, s$	2	2

Table 4: The U formulas and R formulas for the nodes of the Nodeset in Figure 14.

Although the translator assigns three levels to the Büchi automaton (because the original formula has two U subformulas), the second level will be redundant and all the nodes in this level will be unreachable. Therefore, to simplify the graph representations, this level will not be shown.

If all the nodes in the Nodeset have been inserted in the first level of the Büchi automaton, the outgoing transitions of node s_0 can be computed as follows: The four outgoing transitions of s_0 in Figure 14 are s_1, s_2, s_3 and s_4 . The outgoing transitions from nodes s_0 are therefore added with the following nodes as target nodes respectively:

- nodes s'_1 and s'_3 in level 1, because neither q nor s is true at these nodes. (The corresponding R formula for each U formula in the Ulist of each of these nodes is not stored in the Rlist of each node respectively.)
- node s''_2 in level 3, because q is true at this node and satisfies the acceptance condition of level 3. (The corresponding R formula of the U formula in the Ulist of this node is stored in its Rlist.)
- node s''_4 in level 3, because at this node s is true and it also satisfies the acceptance

condition of level 3. (The corresponding R formula of the U formula in the Ulist of this node is stored in the Rlist.)

In a similar manner, node s'_1 has outgoing transitions to nodes s'_1 and s''_2 , and node s'_3 has outgoing transitions to nodes s'_3 and s''_4 (see Figure 15).

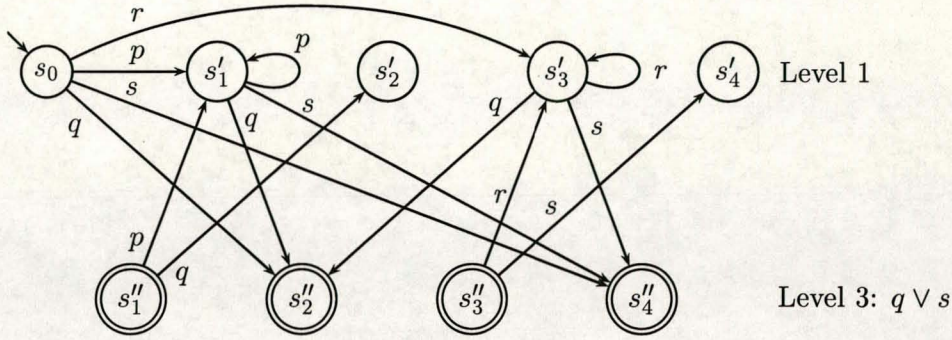


Figure 15: The Büchi automaton for the LTL formula $((p U q) \vee (r U s))$.

After all the nodes have also been added to the third level, the outgoing transitions for s''_1, s''_2, s''_3 and s''_4 are inserted. Nodes s''_2 and s''_4 do not have outgoing transitions and therefore have implicit self loops labelled true. The outgoing transitions of s_1 in Figure 14 are s_1 and s_2 . Because s''_1 is in level 3, its outgoing transitions are added with nodes s'_1 and s'_2 as target nodes. For similar reasons the outgoing transitions of s''_3 are added with nodes s'_3 and s'_4 as target nodes. Nodes s''_1, s''_3, s'_2 , and s'_4 in Figure 15 are unreachable and deleted. The only duplicates are s''_2 and s''_4 and after deletion of s''_4 the final Büchi automaton is the one shown in Figure 16.

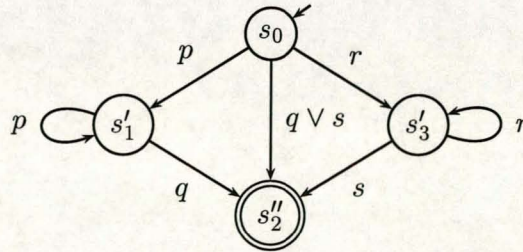


Figure 16: The optimized Büchi automaton for the LTL formula $((p U q) \vee (r U s))$.

Chapter 4

The model checker and its components

The model checker has three components: the compiler, the state generator and the analyser. The compiler translates an ESML specification into a suitable internal form that is interpreted by the state generator to generate new states. The analyser executes the model checking algorithm to verify that a given ESML model satisfies a correctness claim that is expressed as an LTL formula.

4.1 ESML

ESML is a specification language similar to Promela [16]. It supports concurrent processes and is loosely based on CSP [15]. Processes exchange messages by means of communication commands that operate on channels. A simple ESML model of the Alternating Bit protocol is shown in Listing 4.1.

Four message types (a0, a1, d0, and d1) are defined in line 3 of Listing 4.1. These messages can be sent across channels rq and sq, defined in line 7. The main process activates two concurrent processes: process Sender and process Receiver. The processes communicate via the two communication channels, rq and sq. Channels can only pass, and not store, messages. Communication between the Sender and Receiver is synchronous. The Sender alternates between sending message d0 to and receiving acknowledgement a0 from the Receiver, and sending message d1 to and receiving acknowledgement a1 from the Receiver. The Receiver

```

1  MODEL AltBit;
2  TYPE
3      mtype = {a0,a1,d0,d1};
4      int = 0..1;

6  VAR
7      rq, sq: mtype;

9  PROCESS Sender(IN rq: mtype; OUT sq: mtype);
10 VAR
11     s: int;
12 BEGIN
13     s := 0;
14     DO s = 0 → sq!d0; rq?a0; s := 1
15     [] s = 1 → sq!d1; rq?a1; s := 0
16 END
17 END

19 PROCESS Receiver(IN sq: mtype; OUT rq: mtype);
20 VAR
21     s: int;
22 BEGIN
23     s := 0;
24     DO s = 0 → sq?d0; rq!a0; s := 1
25     [] s = 1 → sq?d1; rq!a1; s := 0
26 END
27 END Receiver;

29 BEGIN
30     Sender(rq,sq);
31     Receiver(sq,rq)
32 END AltBit;

34 ASSERT
35     AG ((Sender.s = 0) ⇒ AF(Sender.s = 1))

```

Listing 4.1: An ESMML model for the Alternating Bit Protocol.

alternates between receiving message d0 from and sending acknowledgement a0 to the Sender, and receiving message d1 from and sending acknowledgement a1 to the Sender.

In the parameter list of process Sender, rq is defined as the input channel (a channel on which messages are received) and sq as the output channel (a channel on which messages are sent). The DO command (lines 14–16) executes until all its guards are false. While one or more guards are satisfied, a true guard is selected nondeterministically, and its action is executed. When the command sq!d0 (line 14) is executed, message d0 is sent on channel sq and when rq?a0 (line 14) is executed, message a0 is received on channel rq.

In line 35 a correctness property is given in CTL. This property states that it is always the case that if the Sender sends message d0 it will eventually receive acknowledgement a0. A different approach was chosen for the LTL model checker. Instead of specifying the correctness property at the end of the ESML specification, the user defines propositions. The propositions are then used to specify an LTL formula which is stored in a different file, parsed last, and translated into a Büchi automaton. This allows different LTL formulas, specified using the defined propositions, to be checked without recompiling the ESML model.

For example, two propositions for the ESML model in Listing 4.1 are defined in Listing 4.2. Proposition p is defined to be true when variable s of process Sender is equal to zero and q is defined to be true when variable s of process Sender is equal to 1.

```

34  DEFINITIONS
35       $p : (\text{Sender}.s = 0)$ 
36       $q : (\text{Sender}.s = 1)$ 

```

Listing 4.2: The definitions of propositions p and q .

Keyword DEFINITIONS replaces keyword ASSERT (line 34 of Listings 4.2 and 4.1) and the definitions of p and q replace the CTL formula. The same correctness property as specified in CTL can now be specified in LTL as follows: $G(p \Rightarrow Fq)$.

Further details of the ESML language and compiler fall beyond the scope of this thesis, but the interested reader is referred to [8]. In the next section, the state generator is explained. It builds the state graph of the ESML model on demand, by generating new states as they are requested by the analyser.

4.2 The state generator

The state generator builds the state graph of the ESML model, and is responsible for the cache, stack, and the detection of visited states. The details of the state generator are not relevant here.

Only one data structure was modified to accommodate LTL – a global variable called the state vector. This variable consists of a sequence of bits that represents the combined values of all the local variables and location counters of processes. For the LTL model checker 6 bits were added to the state vector. These six bits are used to store the Büchi automaton state. The state generator receives these six bits from the analyser which implements the

algorithm that computes the product of the state graph and Büchi automaton. The state vector therefore represents the current state of this product.

The state generator interacts with the analyser via two routines: *Execute* and *Backtrack*. *Execute* attempts to generate a new state and *Backtrack* falls back to the current state's predecessor. The analyser alternates between a transition in the Büchi automaton and a transition in the system state graph. The first transition is always executed in the Büchi automaton by the analyser which sends the Büchi automaton state to the state generator. Procedure *Execute* makes a transition in the state graph and updates the state vector to contain the current state of the product state graph. Procedure *Execute* then returns one of the following results to the analyser:

- *Forward* means that a transition in the state graph has been executed successfully.
- *AllChildrenExplored* indicates that all the successors of the current state have been explored.
- *Loop* means that a cycle has been detected. Note that this is a cycle to a state that is a combination of the Büchi automaton state and the system state.
- *Revisit* means that a previously visited state has been reached again. This state is also a combination of the Büchi automaton state and the system state.
- *Complete* is returned when all the successors of the initial system state have been explored.
- *Error* is returned when the execution of a transition in the state graph has led to a runtime failure.

When all the successors of a state have been explored the analyser calls procedure *Backtrack* of the state generator. This procedure returns the current Büchi automaton state to the analyser. A description of the analyser is given in the rest of this chapter.

4.3 The analyser

To check temporal properties, every state transition in the state graph of the original system must be matched with a state transition in the Büchi automaton that represents the temporal property. The goal is to find cycles through acceptance states. Courcoubetis, Vardi, Wolper

and Yannakakis developed such an algorithm [7]. They showed that a nested depth-first search can be used to find accepting states that are reachable from themselves. Listing 4.3 shows a nested depth-first search algorithm where VisitedStates (lines 2 and 10) is a data structure that keeps track of all states already visited during the search. The algorithm works as follows: when the first search backtracks to an accepting state, a second search is started to look for a cycle through this state (line 6).

```

1  PROCEDURE DepthFirstSearch(state s)
2    Add (s,0) to VisitedStates;
3    FOR each successor t of s DO
4      IF (t,0) not in VisitedStates THEN DepthFirstSearch(t) END
5    END
6    IF s is an accepting state THEN seed := s; DepthFirstSearch2(s) END
7  END DepthFirstSearch;

9  PROCEDURE DepthFirstSearch2(state s)
10   Add (s,1) to VisitedStates;
11   FOR each successor t of s DO
12     IF (t,1) not in VisitedStates THEN DepthFirstSearch2(t) END
13     ELSIF t = seed THEN RETURN NotEmpty END
14   END
15 END DepthFirstSearch2;

```

Listing 4.3: The nested depth first search algorithm.

The analyser implements an iterative version of this algorithm. The model checker starts in the initial states of the system and Büchi automaton. In a loop the algorithm then alternates between a transition in the Büchi automaton and a transition in the system state graph.

The analyser will start a second search if all the transitions of a product state have been explored and the current state is an accepting state. If the second search is successful and a cycle through an accepting state is found, the model checking algorithm stops and the analyser returns NotEmpty. The cycle represents a counter example which can be used to find the error. If the second search is not successful, the analyser will backtrack. It also backtracks when all the transitions of a product state have been explored and the state is not accepting. When the analyser has backtracked to the initial state of the state graph and Büchi automaton, the model checking algorithm stops and the analyser returns Empty. This means that the undesirable behaviour, expressed by the negated correctness property cannot be realized from the initial state of the system. The first of the six bits in the state vector allocated to the Büchi automaton state is used as a flag that is set for states visited during the second search. In the worst case, the size of the state space is multiplied by the number

of reachable states of the Büchi automaton.

Implementation details of the iterative nested depth first search algorithm are given in the next section.

4.4 The iterative version of the model checking algorithm

Procedure Check implements the nested depth first search. The algorithm starts by attempting to execute a transition in the Büchi automaton. It then enters a loop that alternates between a transition in the system state graph and a transition in the Büchi automaton. The outline of Procedure Check is shown in Listing 4.4. The code for *a, b, c, d* and *e* is given in Listing A.2 of Appendix A and will be described later in this section.

```

1  PROCEDURE Check;
2  BEGIN
3      r := NextBüchiState(BS);           Execute a transition in the Büchi automaton
4      CASE r OF
5          AcceptEndState: RETURN NotEmpty
6          | NoMove: RETURN Empty
7      END;

9      LOOP
10         res := StateGenerator.Execute;   Execute a transition in the state graph

12         IF res = Complete THEN
13             -- 17      (* do a *)
14         ELSIF (res = Revisit) OR (res = Loop) THEN
15             -- 21      (* do b *)
16         ELSIF (res = Forward) THEN
17             -- 27      (* do c *)
18         ELSE
19             -- 49      (* do d *)
20             END
21             res = AllChildrenExplored
22         IF back THEN
23             -- 61      (* do e *)
24             END
25         END
26         END Check;

```

Listing 4.4: The outline of procedure Check.

The procedure that is called to execute a transition in the Büchi automaton is NextBüchiState which returns one of three values (line 3 of Listing 4.4):

- AcceptEndState: This value is returned when the Büchi automaton state reached is an

accepting state with an implicit self loop labelled true.

- NoMove: The analyser could not execute a transition in the Büchi automaton from the current product state.
- Move: In this case the analyser could execute a transition in the Büchi automaton and the computation of the product can continue.

The loop starts with a call to procedure `Execute` of the state generator and the analyser acts as follows on each return value received:

- Complete: This is returned when all successors of the initial system state have been explored and `a` in Listing 4.4 is executed. When all the successors of the initial Büchi automaton state have also been explored, no counter example could be found and the analyser returns empty. Otherwise computation of the product continues with the next Büchi automaton state. The code that implements `a` is shown in lines (13–17) of Listing A.2 in Appendix A.
- Revisit or Loop: In this case `b` in Listing 4.4 is executed. When the state at which a second search started is revisited and the analyser is still busy with that second search, it means a cycle has been detected through an accepting state and the analyser returns `NotEmpty`. When a loop is detected and the current state is an accepting state, a counter example has been found. In this way a counter example can be detected during the first search, and `NotEmpty` is returned. If `Revisit or Loop` is returned and none of the above conditions apply, the analyser continues execution at the start of the loop where it calls procedure `Execute` to try the next transition in the system state graph. The code that implements `b` is shown in lines (19–21) of Listing A.2 in Appendix A.
- Forward: When this is returned `c` in Listing 4.4 is executed. In this case a new state has been reached and the computation of the product should continue. A transition to the next Büchi automaton state is attempted and if unsuccessful, the backtrack flag is set. If a successful transition was possible and an `AcceptEndState` was reached, `NotEmpty` is returned, otherwise execution continues at the start of the Loop. The code that implements `c` is shown in lines (23–27) of Listing A.2 in Appendix A.
- AllChildrenExplored: There are several possible reactions to this result and the code that implements `d` of Listing 4.4 is shown in Listing 4.5:

```

29      IF system state has no children THEN
30          IF accepting THEN RETURN NotEmpty           Special case
31          ELSE back := TRUE END
32      ELSE
33          r := NextBüchiState(BS);
34          CASE r OF
35              AcceptEndState:
36                  RETURN NotEmpty
37              | Move:
38                  ResetSystemState                     to execute first transition again
39              | NoMove:
40                  IF (search2 AND hitstoredstate) THEN All children of product state explored
41                      Restore values of states;         stop 2nd search
42                      search2 := FALSE;
43                      back := TRUE
44                  ELSE
45                      back := TRUE;
46                      CheckAccepting := TRUE
47                  END
48          END
49      END

```

Listing 4.5: The code that implements the reaction to AllChildrenExplored.

- When a state in the system state graph has no children, but a self loop would result in a counter example, the algorithm stops and returns NotEmpty, otherwise the backtrack flag is set (lines 29–31 of Listing 4.5).
- If all successors of the current system state have been explored, but not all successors of the current Büchi automaton state, the analyser continues with the computation of the product (lines 37–38 of Listing 4.5).
- If all successors of the current state in the product state graph have been explored, the analyser is busy with a second search, and is back at the start state of the second search, the second search is stopped; the state vector is reset and the analyser continues the computation of the product where it stopped to start with the second search (lines 39–43 of Listing 4.5).
- If all successors of the current state in the product state graph have been explored, and the analyser is not busy with a second search, the backtrack and CheckAccept flag is set (lines 45–46 of Listing 4.5). If the CheckAccept flag is set the algorithm checks whether it should start with a second search or backtrack.

The code that implements *e* of Listing 4.4, the actions of the algorithm when the backtrack flag is set, is shown in Listing 4.6. When the analyser is not busy with a second search, the


```

51     IF back THEN
52         back := FALSE;
53         IF (CheckAccepting) & (State = accepting) & (search2 = FALSE) THEN
54             Store values of states;                                Start 2nd search
55             Reset transitions to start at first transition of product state;
56             search2 := TRUE;
57             r := NextBüchiState(BS);
58         ELSE
59             StateGenerator.Backtrack(BS,child);
60             r := BuchiState(BS)                                Try next system state with same Büchi transition
61         END
62     END

```

Listing 4.6: Backtracking in the state graph.

CheckAccept flag is set, and the current state is an accepting state, the analyser starts with a second search. Otherwise it backtracks and continues with the computation of the product.

Chapter 5

Evaluation and conclusions

The goal of the project described in this thesis was to modify an existing model checker for CTL to accept LTL formulas. An automata-theoretic approach was taken where the LTL formulas are first translated into nondeterministic Büchi automata and verification proceeds on-the-fly using a nested depth-first search. After implementation of the LTL model checker its correctness was tested by comparing it against the CTL model checker.

Additional comparisons with SPIN was possible by translating ESML models into Promela and comparing the verification results for various LTL formulas. As an example an alternating bit protocol model specified in both ESML and Promela is given in Figure 17.

The nested depth first search algorithm implemented in SPIN differs from the nested depth first search algorithm implemented in the LTL model checker for ESML. They implement different methods to detect cycles through accepting states during a second search. Therefore a direct comparison between the product state graph generated with SPIN and the product state graph generated with the LTL model checker for ESML was not possible. To test the analyser, a driver module was implemented that accepts simple state graphs. This technique was used to test the behaviour of the LTL model checker for ESML. For comparisons between the CTL and LTL model checkers, the same ESML model could be used. Testing included checking that both the LTL and CTL model checker developed at Stellenbosch generate the same number of unique states for the same models.

The translator implemented to translate LTL formulas to Büchi automata is compared to other techniques in Section 5.1. Efficiency issues of the model checking algorithm are discussed in Section 5.2. Future work are discussed in Section 5.3 and some general conclusions are

<pre> 1 MODEL AltBit; 2 TYPE 3 mtype = {a0,a1,d0,d1}; 4 int = 0..1; 6 VAR 7 rq, sq: mtype; 9 PROCESS Sender(IN in: mtype; OUT out: mtype); 10 VAR 11 s: int; 12 BEGIN 13 s := 0; 14 DO s = 0 → out!d0; in?a0; s := 1 15 [] s = 1 → out!d1; in?a1; s := 0 16 END 17 END 19 PROCESS Receiver(IN in: mtype; OUT out: mtype); 20 VAR 21 s: int; 22 BEGIN 23 s := 0; 24 DO s = 0 → in?d0; out!a0; s := 1 25 [] s = 1 → in?d1; out!a1; s := 0 26 END 27 END Receiver; 29 BEGIN 30 Sender(rq,sq); 31 Receiver(sq,rq) 32 END AltBit; </pre>	<pre> 1 mtype = {a0,a1,d0,d1}; 3 proctype Sender(chan in,out); 4 { 5 bit s; 6 s = 0; 7 do 8 :: s == 0 → out!d0; in?a0; s = 1; 9 :: s == 1 → out!d1; in?a1; s = 0; 10 od 11 } 13 proctype Receiver(chan in,out); 14 { 15 bit s; 16 s = 0; 17 do 18 :: s == 0 → in?d0; out!a0; s = 1; 19 :: s == 1 → in?d1; out!a1; s = 0; 20 od 21 } 23 init 24 { 25 chan rq = [0] of mtype 26 chan sq = [0] of mtype 28 run Sender(rq,sq); 29 run Receiver(sq,rq) 30 } </pre>
--	---

Figure 17: An ESML (left) and a Promela (right) model for the alternating bit protocol

given in Section 5.4.

5.1 LTL to Büchi automata translator

The translator for LTL formulas into Büchi automata was implemented based on the on-the-fly algorithm developed by Gerth, Peled, Vardi and Wolper (GPVW) [14] and algorithms implemented in SPIN [16]. The translator was implemented to build the Büchi automaton before the model checking procedure. The GPVW algorithm was used to implement the first part of the translator that generates a graph representing a generalized Büchi automaton from an LTL formula. SPIN code was analysed and used to implement the second part of the translator which builds a simple Büchi automaton from the graph and optimize it by reducing the number of nodes of the Büchi automaton.

The GPVW algorithm is an improvement on the global construction technique described in [31]. The global construction technique generates a node for each set of subformulas of the original formula, which leads to the worst case exponential complexity in space. Büchi

automata generated from the GPVW algorithm have fewer nodes than those generated with the global construction. The translator described in this thesis builds the Büchi automaton before model checking starts. As a result an optimization algorithm could be implemented to generate Büchi automata with still fewer nodes than those generated by the GPVW algorithm. The translator will be referred to as the optimizing translator in the rest of this section.

LTL formula	Global	GPVW	Optimizing translator
$p \cup q$	8	3	2
$GFp \Rightarrow GFq$	114	9	6
$Fp \cup Gq$	56	8	5
$\neg((FFp \Rightarrow Fp) \wedge (Fp \Rightarrow FFp))$	–	22	3

Table 5: The number of nodes in Büchi automata generated with three different translation algorithms.

The results of a comparison between the global construction technique, the GPVW algorithm and the optimizing translator are shown in Table 5. For each algorithm the number of nodes in the final Büchi automaton is given for each of four different LTL formulas. The last column shows the results of the optimizing translator and the first three columns are from [14]. For the second and third LTL formulas, the Büchi automata generated with the optimizing translator have 3 fewer nodes than the Büchi automata generated with the GPVW algorithm, and 1 fewer node for the first example in the table. For the last formula in the table the optimizations implemented in the optimizing translator resulted in a useful reduction in size of the Büchi automaton compared to the one generated by the GPVW algorithm. (There was insufficient memory to complete the global construction of the Büchi automaton for this last example.)

Although it takes extra time to optimize the Büchi automaton, the extra time is small compared to the overall construction of the automaton; and because the automaton is built before the model checking procedure, the memory used and time taken to build the automaton is less significant than the size of the final Büchi automaton. A small Büchi automaton is important, because the emptiness check takes time (linearly) proportional to the size of the constructed automaton. The advantage of the optimizing translator is that it results in a smaller automaton which saves time searching for counter examples during state exploration and leads to shorter error trails.

5.2 The model checking algorithm

The model checking algorithm implemented is based on an algorithm developed by Courcoubetis, Vardi, Wolper and Yannakakis [7]. It builds the system state graph and does the model checking on-the-fly using a nested depth-first search. The outer search is called the first search and the search called inside the outer search is called the second search. The second search is called when the first search has backtracked to an accepting state. The purpose of the second search is to find cycles through accepting states. In the worst case, the time might double if all the states are reachable during both searches and there are no cycles through accepting states. Typically, however, fewer states are visited during the second search than during the first search, as is the case for the examples in Table 6. In all these examples the number of states visited during the second search is less than 30% of the number of states visited during the first search.

	Model	First Search (x)	Second Search (y)	y as a % of x	Result
1	ME	172	46	26.70	satisfied
2	ME	9	0	0.00	violated
3	ME	33	4	12.00	violated
4	PC	6252	1607	25.70	satisfied
5	PC	173	0	0.00	violated
6	DP	552	96	17.40	satisfied
7	DP	42	0	0.00	violated
8	DP	97	6	6.20	violated
9	SW	127472	2677	2.10	satisfied
10	SW	7112	68	0.96	violated

Table 6: Second search results. Columns 3 and 4 show the number of states visited during each search.

The models verified were mutual exclusion (ME), a producer-consumer example (PC), dining philosophers (DP) and the sliding window protocol (SW). Different versions of the models were checked, some with errors and others without. The result of the verification is shown in the last column. The number of states generated during the first and second searches is shown in the second and third columns respectively. The models were verified against liveness properties of the form $G(p \Rightarrow Fq)$.

In three cases a counter example was found without performing a second search. This happens because our algorithm stops when a cycle is found at an accepting state. An example of such

a case would be when state s_2 in Figure 18a is an accepting state. This saves time, because a counter example can then be returned without performing a second search.

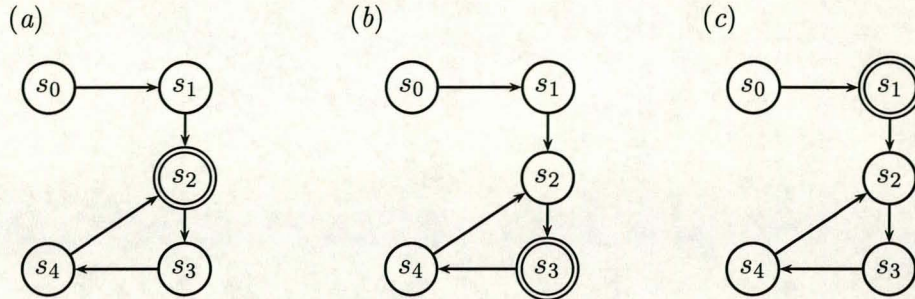


Figure 18: Executing the nested depth-first search on a graph with one cycle.

Another change to the nested depth-first search that could increase its efficiency, would be to stop the first search when a cycle through an accepting state is found. For example, if state s_3 in Figure 18b is an accepting state and the cycle at state s_2 is found during the first search. For this to work, enough information about the positions of acceptance states in the current path should be stored so that, if state s_1 in Figure 18c, for example, is an accepting state, finding a cycle at node s_2 would not result in an incorrect termination of the first search.

Another observation from Table 6, is that fewer states are generated in an attempt to verify a model which contains an error than for models that satisfy the correctness property. The reason is that the algorithm implemented generate only the subset of the graph needed to compute the result of the verification, and that errors occur on different search depths (see rows 2 and 3, and rows 7 and 8 in Table 6). The results in the table show that errors are usually found before the entire state graph has been generated.

If we compare this nested depth-first search algorithm to the SCC (strongly connected component) detection of the model checking algorithm in the CTL model checker developed previously at Stellenbosch, we find that each favour one of two tradeoffs, namely time or space. An SCC is a component of a graph in which all the states are reachable from all the other states in the component. It was found that SCC detection is not expensive in terms of runtime, but memory requirements, on the other hand, increase dramatically for some models. In the Sliding Window Protocol, for example, the largest SCC is only 0.8% of the unique states whereas the largest SCC for the Dining Philosophers model is 57.13% of

its unique states. The reason why the detection of SCCs places a high demand on memory requirements, is that the entire SCC must be stored in memory during its construction.

With the nested depth-first search one path at a time is investigated. It therefore has a better memory usage, but because of its second search, it takes longer. We argue that optimizing memory usage is more important than optimizing speed and that for this reason, the nested depth-first search is preferable over the detection of SCCs.

5.3 Future work

During the testing phase it became apparent that most of the models verified contained unfair execution paths, specifically paths that are not strongly fair. Strong fairness requirements are difficult to guarantee, because by expressing strong fairness in the LTL formula some errors can be missed. As an example, consider the mutual exclusion model of which the state graph is shown in Figure 19.

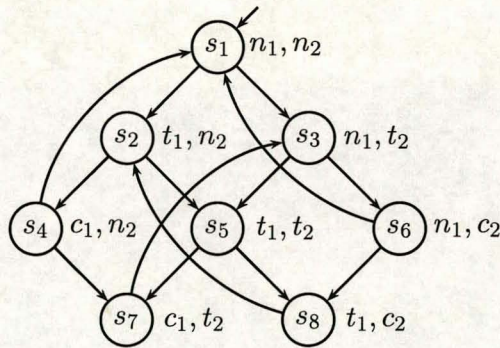


Figure 19: The state graph of a mutual exclusion model.

When the mutual exclusion model is checked against $G(t_1 \Rightarrow F(c_1))$, it is found that an infinite cycle through states s_2, s_5 and s_8 violates the property. This path is not strongly fair, because the transitions to c_1 are enabled infinitely often, but not executed infinitely often.

If $G(t_1 \Rightarrow F(c_1))$ is changed to $G(t_1 \Rightarrow (F(c_1) \vee GF(c_2)))$ to express the fact that the cycle through s_2, s_5 and s_8 is possible, but should not be given as an error, the model checker returns *satisfied*.

However, checking the state graph in Figure 20 against the formula $G(t_1 \Rightarrow (F(c_1) \vee GF(c_2)))$,

the model checker will also return *satisfied*. This is clearly wrong. The error will be found by checking whether there exists a path on which F_{c_1} is true. The model checker does this by building the Büchi automaton for F_{c_1} (instead of $\neg F_{c_1}$) and returning satisfied if the product of the Büchi automaton and state graph is not empty.

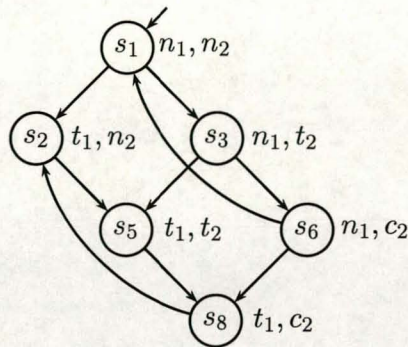


Figure 20: The state graph of a mutual exclusion model with an error.

Finding errors that were missed by the formula expressing strong fairness is often difficult and the preferred solution is to rewrite the model instead of the formula. Future work should be to implement different forms of fairness in the model checker and compare the results with rewriting the LTL formula to express fairness.

Future work includes improvements in the model checking algorithm itself. The changes to the nested depth-first search algorithm, suggested in the previous section, can be implemented and compared against the current implementation. The current implementation is based on the original algorithm in [7], but Holzmann, Peled and Yannakakis [17] revised the standard algorithm to be compatible with partial orders. This revised algorithm proved to be more efficient, even for verification without partial orders. The improvement comes from the fact that the second search stops when a cycle to a state visited during the first search is found in the current path. This is easy to implement and it will be worthwhile to make the changes explained in [17] to the implementation of the LTL model checker for ESML. As a result, experimentation with partial orders would also be possible.

5.4 General conclusions

The experimental model checker for LTL is an improvement over the CTL model checker in several respects. First, LTL is more useful in practice than CTL. There are correctness claims that can be expressed in LTL that cannot be expressed in CTL, for example FGp . This formula can be used to check that a condition holds infinitely often on a path, but not necessarily at every state on the path. LTL can also be used to express strong fairness, whereas CTL cannot. For example the LTL formula $GFp \Rightarrow GFq$ can be used to check that a path is strongly fair in terms of a transition that is enabled (p) and the execution of the transition (q). Although there are also correctness claims expressible in CTL that cannot be expressed in LTL, such as $AG(p \Rightarrow EFq)$, they are seldom useful.

Second, the model checking algorithm of the LTL model checker has better memory usage than the CTL model checker. The model checking algorithm of the LTL model checker uses a second search as an alternative to building SCCs. Both methods are used to detect cycles in a state graph. SCC detection is memory intensive, because the entire SCC must be stored in memory during its construction. However, most of the time the SCCs are not very big and therefore the advantage in memory usage of the second search is not meaningful. But it is possible that the biggest part of a very large state graph is an SCC and in such a case the advantage in memory will be meaningful.

CTL model checking has an advantage in time over LTL model checking. The time complexity of CTL model checking is linear in the number of reachable states and the length of the formula. On the other hand, the time complexity of LTL model checking is linear in the number of reachable states, but exponential in the length of the formula. However, LTL formulas are usually short in practice and therefore the advantage in time of CTL over LTL is doubtful.

Appendix A

Algorithms

```

0  MODULE CreateGraph;
1  TYPE
2  NodeType = RECORD
3      Name: String;
4      Incoming: Set of String;
5      Outgoing: Set of String;
6      New: Set of Formula;
7      TFormulas: Set of Formula;
8      Next: Set of Formula;
9  END;

11 NodesetPtr = POINTER TO NodesetType;
12 NodesetType = RECORD
13     Node: NodeType;
14     Ulist, Rlist: List of Integer;
15     Next: NodesetPtr;
16 END;

18 VAR
19     Nodeset, Nodestack: NodesetPtr;
20     Node, tmp: NodeType;

22 PROCEDURE Expand(Node: NodeType);
23 BEGIN
24     IF Node.New =  $\emptyset$  THEN
25         IF  $\exists$ node  $\in$  Nodeset with
26             node.TFormulas = Node.TFormulas and node.Next = Node.Next THEN
27             node.Incoming := node.Incoming  $\cup$  Node.Incoming
28         ELSE

```

data structure of a node

formulas to be expanded

formulas true at the current node

formulas that must be true from the next node on

check whether New is empty

check for duplicate node in Nodeset

```

29  AddToSet(Nodeset,Node);
30  IF Node.Next  $\neq \emptyset$  THEN
31      Node1.Incoming := Node.Name;
32      Node1.Outgoing :=  $\emptyset$ ;
33      Node1.Name := newname();
34      Node1.New := Node.Next;
35      Node1.TFormulas :=  $\emptyset$ ;
36      Node1.Next :=  $\emptyset$ ;
37      PushOnStack(Nodestack,Node1);
38  END
39  END
40  ELSE
41      let  $\phi \in \text{New}$ ;
42      Node.New := Node.New \  $\{\phi\}$ ;
43      IF Proposition( $\phi$ ) and ( $\phi = \text{false}$  or  $\neg\phi \in \text{Node.TFormulas}$ ) THEN
44          (*Discard current node*)
45      ELSIF Proposition( $\phi$ ) THEN
46          Node.TFormulas := Node.TFormulas  $\cup \{\phi\}$ ;
47          AddToSet(Nodeset,Node);
48      ELSIF  $\phi = p \vee q$ , or  $\phi = p \cup q$ , or  $\phi = p \vee q$ 
49          Node1.Name := newname();
50          Node2.Name := newname();
51          Node1.Incoming := Node.Incoming;
52          Node2.Incoming := Node.Incoming;
53          Node1.Outgoing :=  $\emptyset$ ;
54          Node2.Outgoing :=  $\emptyset$ ;
55          Node1.TFormulas := Node.TFormulas  $\cup \{\phi\}$ ;
56          Node2.TFormulas := Node.TFormulas  $\cup \{\phi\}$ ;
57          IF  $\phi = p \vee q$  THEN
58              Node1.New := Node.New  $\cup (p \setminus \text{Node.TFormulas})$ ;
59              Node1.Next := Node.Next;
60              Node2.New := Node.New  $\cup (q \setminus \text{Node.TFormulas})$ ;
61              Node2.Next := Node.Next;
62          ELSIF  $\phi = p \cup q$  THEN
63              Node1.New := Node.New  $\cup (p \setminus \text{Node.TFormulas})$ ;
64              Node1.Next := Node.Next  $\cup (p \cup q)$ ;
65              Node2.New := Node.New  $\cup (q \setminus \text{Node.TFormulas})$ ;
66              Node2.Next := Node.Next;
67          ELSE

```

add node to Nodeset

create new node and push on stack

remove formula ϕ from New

node fully expanded


```

68     Node1.New := Node.New  $\cup$   $((p \wedge q) \setminus \text{Node.TFormulas})$ ;
69     Node1.Next := Node.Next;
70     Node2.New := Node.New  $\cup$   $(q \setminus \text{Node.TFormulas})$ ;
71     Node2.Next := Node.Next  $\cup$   $(p \vee q)$  ;
72     END
73     PushOnStack(Nodestack,Node1);
74     PushOnStack(Nodestack,Node2);
75     ELSE                                      $\phi = p \wedge q$ : add  $p$  and  $q$  to New if not in New yet
76     Node.New := Node.New  $\cup$   $(\{p,q\} \setminus \text{Node.TFormulas})$ ;
77     Node.Next := Node.Next;
78     PushOnStack(Nodestack,Node);
79     END
80     END
81     END Expand;

83     BEGIN
84     Nodeset := new(NodesetType);
85     Nodestack := new(NodesetType);
86     Node.Name := newname();
87     Node.Incoming := {init};
88     Node.Outgoing :=  $\emptyset$ ;
89     Node.New := Original formula;
90     Node.TFormulas :=  $\emptyset$ ;
91     Node.Next :=  $\emptyset$ ;
92     PushOnStack(Nodestack,Node);

94     WHILE Nodestack  $\neq \emptyset$  DO
95         tmp := PopOffStack(Nodestack);
96         Expand(tmp);
97     END

99     Node.Name := init;
100    Node.Incoming :=  $\emptyset$ ;
101    Node.Outgoing :=  $\emptyset$ ;
102    Node.New :=  $\emptyset$ ;
103    Node.TFormulas := Original formula;
104    Node.Next :=  $\emptyset$ ;
105    AddToSet(Nodeset,Node);

107    END CreateGraph.

```

*initialise the start node, this is not
the initial node of the Büchi automaton*

push start node on stack

expand all nodes on Nodestack

Add the initial node of the Büchi automaton

Listing A.1: The expansion algorithm.


```

29      IF system state has no children THEN
30          IF accepting THEN RETURN NotEmpty Special case
31          ELSE back := TRUE END
32      ELSE
33          r := NextBüchiState(BS);
34          CASE r OF
35              AcceptEndState:
36                  RETURN NotEmpty
37              | Move:
38                  ResetSystemState to execute first transition again
39              | NoMove: All children of product state explored
40                  IF (search2 AND hitstoredstate) THEN
41                      Restore values of states; stop 2nd search
42                      search2 := FALSE;
43                      back := TRUE
44                  ELSE
45                      back := TRUE;
46                      CheckAccepting := TRUE
47                  END
48          END
49      END
50  END
51  IF back THEN
52      back := FALSE;
53      IF (CheckAccepting) & (State = accepting) & (search2 = FALSE) THEN
54          Store values of states; Start 2nd search
55          Reset transitions to start at first transition of product state;
56          search2 := TRUE;
57          r := NextBüchiState(BS);
58      ELSE
59          StateGenerator.Backtrack(BS,child);
60          r := BüchiState(BS) Try next system state with same Büchi transition
61      END
62  END
63  END
64  END Check;

```

Listing A.2: The procedure that runs the model checking algorithm.

Bibliography

- [1] C.C. Ackerman. Providing mechanical support for program development in a weakest precondition calculus. Master's thesis, University of Stellenbosch, April 1993.
- [2] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Proceedings of the 6th International Conference on Computer-Aided Verificaton*, volume 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
- [3] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *Proceedings of the 10th Symposium on Logic in Computer Science*, San Diego, California, June 1995.
- [4] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method and Philosophy of Science*, pages 1–12, Stanford, 1960.
- [5] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. 1981.
- [6] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
- [7] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

- [8] P.J.A. de Villiers and W. Visser. ESML—a validation language for concurrent systems. In Judy Bishop, editor, *7th Southern African Computer Symposium*, pages 59–64, July 1992.
- [9] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science, 1990.
- [10] E.A. Emerson. Automated temporal reasoning about reactive systems. In *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 41–92. Springer-Verlag, 1996.
- [11] E.A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. In *Science of Computer Programming*, volume 8, pages 275–306. Elsevier Science Publishers B.V. (North-Holland), 1987.
- [12] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jéron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1:251–273, 1992.
- [13] M. Fisher. A model checker for linear time temporal logic. *Formal Aspects of Computing*, 4(3):299–319, 1992.
- [14] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th Workshop on Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.
- [15] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, New Jersey, 1985.
- [16] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [17] G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proceedings of the 2nd Workshop in the SPIN Verification System*. American Mathematical Society, DIMACS/39, August 1996.
- [18] O. Kupferman and M.Y. Vardi. Weak alternating automata and tree automata emptiness. In *The 30th Annual ACM Symposium on Theory of Computing*, May 1998.
- [19] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2), March 1977.

- [20] L. Lamport. “sometime” is sometimes “not never”—on the temporal logic of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, Las Vegas, January 1980.
- [21] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
- [22] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.
- [23] D.E. Muller, A. Saoudi, and P.E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings of the 3rd IEEE Symposium on Logic in Computer Science*, pages 422–427, Edinburgh, July 1988.
- [24] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.
- [25] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, Berlin/New York, 1982.
- [26] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.
- [27] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. In *Theoretical Computer Science*, volume 49, pages 217–237. Elsevier Science Publishers B.V. (North-Holland), 1987.
- [28] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Models and Semantics*, volume B, pages 135–191. Elsevier Science, 1990.
- [29] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996.
- [30] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

- [31] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.
- [32] B. Vergauwen and J. Lewi. A linear local model checking algorithm for CTL. In *Proceedings of the 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 447–461. June 1993.
- [33] W.C. Visser. *Efficient CTL* Model Checking Using Games and Automata*. PhD thesis, University of Manchester, June 1998.
- [34] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computations. In *The 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.